FLUIDLAB: A DIFFERENTIABLE ENVIRONMENT FOR BENCHMARKING COMPLEX FLUID MANIPULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Humans manipulate various kinds of fluids in their everyday life: creating latte art, scooping floating objects from water, rolling an ice cream cone, etc. Using robots to augment or replace human labors in these daily settings remain as a challenging task due to the multifaceted complexities of fluids. Previous research in robotic fluid manipulation mostly consider fluids governed by an ideal, Newtonian model in simple task settings (*e.g.*, pouring water into a container). However, the vast majority of real-world fluid systems manifest their complexities in terms of the fluid's complex material behaviors (*e.g.*, elastoplastic deformation) and multi-component interactions (*e.g.* coffee and frothed milk when making latte art), both of which were well beyond the scope of the current literature. To evaluate robot learning algorithms on understanding and interacting with such complex fluid systems, a comprehensive virtual platform with versatile simulation capabilities and well-established tasks is needed. In this work, we introduce *FluidLab*, a simulation environment with a diverse set of manipulation tasks involving complex fluid dynamics. These tasks address interactions between solid and fluid as well as among multiple fluids. At the heart of our platform is a fully differentiable physics simulator, *FluidEngine*, providing GPU-accelerated simulations and gradient calculations for various material types and their couplings, extending the scope of the existing differentiable simulation engines. We identify several challenges for fluid manipulation learning by evaluating a set of reinforcement learning and trajectory optimization methods on our platform. To address these challenges, we propose several domain-specific optimization schemes coupled with differentiable physics, which are empirically shown to be effective in tackling optimization problems featured by fluid system's non-convex and non-smooth properties. FluidLab and FluidEngine will be publicly available. See our project page[1] for more demos.

1 INTRODUCTION

Imagine you are fishing on the lakeside. Your hat falls into the water and starts to float out of reach. In order to get it back, you use your hands to paddle the water gently, generating a current that pulls the the hat back into reach. In this scenario, the water functions as a transmitting medium for momentum exchange to accomplish the manipulation task. In fact, from creating latte art to making ice creams, humans frequently interact with different types of fluids everyday (Figure 1). However, performing these tasks are still beyond the reach of today's robotic systems due to three major challenges. First, fluid systems exhibit a wide range of material models (e.g., air, liquid, granular flow, etc. (Bridson, 2015)) that are difficult to simulate within a unified framework. Second, describing the state of a fluid system under manipulation requires a high dimensional state space (Lin et al., 2020). Third, the nonlinear dynamics of different coupling models pose unique challenges to efficient solution search in the state space (Schenck & Fox, 2018). For example, although water and smoke face the same computational challenges in solving fluid equations, manipulating these two mediums in different physical contexts requires different strategies.

Previous research in robotic manipulation covering fluids mostly adopts relatively simple task settings, and usually considers tasks with a single-phase fluid, such as pouring water (Schenck & Fox, 2018; Lin et al., 2020; Sermanet et al., 2018) or scooping objects from water (Seita et al., 2022;

[1]Project Page: https://sites.google.com/view/fluidlab

Figure 1: We encounter various manipulation tasks involving fluids in everyday life. Left: real-world scenarios including making latte art, making ice creams and designing air ventilation systems. Right: simulation tasks in FluidLab involving latte art, ice-creams and indoor air circulation.

Antonova et al., 2022). In this work, we aim to study the problem of *complex fluid manipulation* with more challenging task settings and complex goal configurations.

Since virtual environments have played an instrumental role in benchmarking and developing robot learning algorithms (Brockman et al., 2016; Kolve et al., 2017; Xiang et al., 2020), it is desirable to have a virtual platform providing a versatile set of fluid manipulation tasks. In this work, we propose *FluidLab*, a new simulation environment to pave the way for robotic learning of complex fluid manipulation skills that could be useful for real-world applications. FluidLab provides a new set of manipulation tasks that are inspired by scenarios encountered in our daily life, and consider both interactions within multiple fluids (e.g., coffee and frothed milk), as well as the coupling between fluids and solids. Building such an environment poses a major challenge to the simulation capability: fluids encountered in our life consist of a wide spectrum of materials with different properties; in addition, simulating these tasks also requires modeling interactions with other non-fluid materials. For example, coffee is a type of (mostly inviscid) *Newtonian* liquid; frothed milk behaves as *viscous Newtonian* liquid (i.e., with a constant viscosity that does not change with stress) (Figure 1, R-A), while ice cream is a type of *non-Newtonian* fluid (in particular, a shear-thinning fluid with its viscosity decreasing as the local stress increasing) (Figure 1, R-B); when stirring a cup of coffee, we use a *rigid* stick to manipulate the coffee (Figure 1, R-A). However, there still lacks a simulation engine in the research community that supports such a wide variety of materials and couplings between them. Therefore, we developed our own physics engine, named *FluidEngine*, to support simulating the tasks proposed in FluidLab. FluidEngine is a general-purpose physics simulator that supports modeling solid, liquid and gas, covering materials including elastic, plastic and rigid objects, Newtonian and non-Newtonian liquids, as well as smoke and air. FluidEngine is developed using Taichi (Hu et al., 2019b), a domain-specific language embedded in Python, and supports massive parallelism on GPUs. In addition, it is implemented in a fully *differentiable* manner, providing useful gradient information for downstream optimization tasks. Closest to FluidEngine is Nvidia FleX (Macklin et al., 2014), a physics engine that supports both rigid and deformable objects and is widely used in a variety of manipulation environments (Lin et al., 2020; Gan et al., 2020; Schenck & Fox, 2018). However, FleX doesn't support simulating plastic materials or air, and is not differentiable. FluidEngine follows the OpenAI Gym API (Brockman et al., 2016) and also comes with a user-friendly Python interface for building custom environments. We also provide a GPU-accelerated renderer developed with OpenGL, supporting realistic image rendering in real-time.

We evaluate state-of-the-art Reinforcement Learning algorithms and trajectory optimization methods in FluidLab, and highlight major challenges with existing methods for solving the proposed tasks. In addition, since differentiable physics has proved useful in many rigid-body and soft-body robotic tasks (Xu et al., 2021; Lin et al., 2022a; Xu et al., 2022; Li et al., 2022), it is desirable to extend it to fluid manipulation tasks. However, supporting such a wide spectrum of materials present in FluidLab in a differentiable way is extremely challenging, and optimization using the gradients is also difficult due to the highly non-smooth optimization landscapes of the proposed tasks. We propose several optimization schemes that are generally applicable to the context of fluids, making our simulation fully differentiable and providing useful directions to utilize gradients for downstream optimizations tasks. We demonstrate that our proposed techniques, coupled with differentiable physics, could be effective in solving a set of challenging tasks, even in those with highly turbulent fluid (e.g., inviscid air flow). We summarize our main contributions as follows:

- We propose *FluidLab*, a set of standardized manipulation tasks to help comprehensively study *complex fluid manipulation* with challenging task settings and potential real-world applications.

- We present *FluidEngine*, the underlying physics engine of FluidLab that models various material types required in FluidLab's tasks. To the best of our knowledge, it is the first-of-its-kind physics engine that supports such a wide spectrum of materials and couplings between them, while also being *fully differentiable*.
- We benchmark existing RL and optimization algorithms, highlighting challenges for existing methods, and propose optimization schemes that make differentiable physics useful in solving a set of challenging tasks. We hope FluidLab could benefit future research in developing better methods for solving complex fluid manipulation tasks.

2 Related Work

Robotic manipulation of rigid bodies, soft bodies and fluids Robotic manipulation has been extensively studies over the past decades, while most of previous works concern tasks settings with mainly rigid objects (Zeng et al., 2020; Suárez-Ruiz et al., 2018; Tung et al., 2020; Chen et al., 2022). In recent years, there has been a rising interest in manipulation problems concerning systems with deformable objects, including elastic objects such as cloth (Liang et al., 2019; Wu et al., 2019; Lin et al., 2022b), as well as plastic objects like plasticine (Huang et al., 2021). Some of them also study manipulation problems involving fluids, but mainly consider relatively simple problems settings such as robotic pouring (Lin et al., 2020; Narasimhan et al., 2022) or scooping (Antonova et al., 2022), and mostly deal with only water (Li et al., 2018; Schenck & Fox, 2018; Ma et al., 2018; Seita et al., 2022; Antonova et al., 2022). In this work, we aim to study a comprehensive set of manipulation tasks involving different fluid types with various properties. It is also worth mentioning that aside from the robotics community, a vast literature has been devoted to fluid control in computer graphics (Hu et al., 2019a; Zhu et al., 2011; Yang et al., 2013), with their focus on artistic visual effects yet neglecting the physical feasibility (e.g., arbitrary control forces can be added to deform a fluid body into a target shape). In contrast to them, we study fluid manipulation from a realistic *robotics* perspective, where fluids are manipulated with an embodied robotic agent.

Simulation environments for robotic policy learning In recent years, simulation environments and standardized benchmarks have significantly facilitated research in robotics and policy learning. However, most of them (Brockman et al., 2016; Kolve et al., 2017; Xiang et al., 2020) use either PyBullet (Coumans & Bai, 2016) or MuJoCo (Todorov et al., 2012) as their underlying physics engines, which only supports rigid-body simulation. Recently, environments supporting soft-body manipulation, such as SAPIEN (Xiang et al., 2020), TDW (Gan et al., 2020) and SoftGym (Lin et al., 2020) adopted Nvidia FleX (Macklin et al., 2014) engine to simulate deformable objects and fluids. However, FleX only supports limited types of materials, and its underlying Position-Based Dynamics (PBD) solver has been considered not physically accurate enough to produce faithful interactions between solids and fluids. (See Section 3 for a comparison).

Differentiable physics simulation Machine Learning with differentiable physics has gained increasing attention recently. One line of research focuses on learning physics models with neural networks (Li et al., 2018; Pfaff et al., 2020), which are intrinsically differentiable and could be used for downstream planning and optimization. Learning these models assumes ground-truth physics data provided by simulators, hasn't proved to be capable of simulating a wide range of materials, and could suffer from out-of-domain generalization gap (Sanchez-Gonzalez et al., 2020). Another approach is to implement physics simulations in a differentiable manner, usually with automatic differentiation tools (Hu et al., 2019a; de Avila Belbute-Peres et al., 2018; Xu et al., 2022; Huang et al., 2021), and the gradient information provided by these simulations has shown to be helpful in control and manipulation tasks concerning rigid and soft bodies (Xu et al., 2021; Lin et al., 2022a; Xu et al., 2022; Li et al., 2022; Huang et al., 2021). On a related note, Taichi (Hu et al., 2019b) and Nvidia Warp (Macklin, 2022) are two recently developed domain-specific programming language for GPU-accelerated simulation. FluidLab is implemented using Taichi.

3 FluidEngine

3.1 System Overview

FluidEngine is developed in Python and Taichi, and provides a set of user-friendly APIs for building simulation environments. At a higher level, it follows the standard OpenAI Gym API and is com-

patible with standard reinforcement learning and optimization algorithms. Each environment created using FluidEngine consists of five components: i) a robotic agent equipped with user-defined end-effector(s); ii) objects imported from external meshes and represented as signed distance fields (SDFs); iii) objects created either using shape primitives or external meshes, represented as particles; iv) gas fields (including a velocity field and other advected quantity fields such as smoke density and temperature) for simulating gaseous phenomena on Eulerian grids; and v) a set of user-defined geometric boundaries in support of sparse computations. After adding these components, a complete environment is built by calling a `build` method, which enables inter-component communications for simulation. (See Appendix A for a sample code of a typically environment constructed following this pipeline). Our full source code will be released upon publication. We encourage readers to refer to our project page for visualizations of a diverse set of scenes simulated with FluidEngine.

3.2 Material Models

FluidEngine supports various types of materials, mainly falling under three categories: solid, liquid and gas. For solid materials, we support rigid, elastic and plastic objects. For liquid, we support both Newtonian and non-Newtonian liquids. Both solid and liquid materials are represented as particles, simulated using the Moving Least Squares Material Point Method (MLS-MPM) (Hu et al., 2018), which is a hybrid Lagrangian-Eulerian method that uses particles and grids to model continuum materials. For gas such as smoke or air, we simulate them as incompressible fluid on a Cartersian grid using the advection-projection scheme (Stam, 1999).

Elastic, plastic, and liquid Both elastic and plastic materials are simulated following the original MLS-MPM formulation (Hu et al., 2018). For both materials, we compute the updated deformation gradient, internal force, and material stress in the particle-to-grid (P2G) step, and then enforce boundary conditions in grid operations. Corotated constitutive models are used for both materials. For plastic material, we additionally adopt the Box yield criterion (Stomakhin et al., 2013) to update deformations. We refer readers to the original MPM papers (Hu et al., 2018; Stomakhin et al., 2013) for more details. Both viscous and non-viscous liquids are also modeled using the corotated models, and we additionally reset the deformation gradients to diagonal matrices at each step. For non-Newtonian liquids such as ice cream, we model them using the elastoplastic continuum model with its plastic component treatsd by the von Mises yield criterion (following Huang et al. (2021)).

Rigid body Two types of geometric representations are used to support rigid bodies in FluidEngine. The first type is represented by particles. We simulate rigid objects by treating them as elastic objects first using MLS-MPM, then we compute an object-level transformation matrix and enforce the rigidity constraint. Another type of rigid object is represented using signed distance fields (SDFs), usually generated by importing external meshes. This representation is more computation-efficient since it computes its dynamics as a whole, generally used for meshes with big sizes.

Gas We develop a grid-based incompressible fluid simulator following (Stam, 1999) to model gaseous phenomena such as air and smoke. A particle-based method is not well suited in these cases due to its (weakly) compressible limit.

Inter-material coupling MPM naturally supports coupling between different materials and particles, computed in the grid operations. For interactions (collision contact) between mesh-based objects and particle-based materials, we represent meshes as time-varying SDFs and model the contact by computing surface normals of SDFs and applying Coulomb friction (Stomakhin et al., 2013). For interaction between mesh-based objects and the gas field, we treat grid cells occupied by meshes as the boundaries of the gas field and impose boundary conditions in the grid operations. Coupling between gas and particle-based materials is also implemented at the grid level, by treating grid cells occupied by particles as boundaries.

3.3 Differentiability and Rendering

FluidEngine is implemented based on Taichi's auto diff system to compute gradients for simple operations. We further implement our own gradient computations for complex operations such as SVD and the projection step in the gas simulation, and efficient gradient-checkpointing to allow gradient flow over unlimited time horizons, unconstrained by GPU memory size. FludiEngine also comes with a real-time OpenGL-based renderer. More details are in Appendix B.

3.4 COMPARISON WITH OTHER SIMULATION ENVIRONMENTS

Simulators	Differentiable	Solid			Liquid		Gas
		Rigid	Elastic	Plastic	Newt.	Non-Newt.	Air / Smoke
PyBullet		✓	✓				
MuJoCo		✓					
FleX		✓	✓		✓		✓[B]
SoftGym		✓	✓		✓		
PlasticineLab	✓[A]	✓		✓			
SAPIEN		✓	✓		✓		
TDW		✓	✓				
DEDO		✓	✓				
FluidLab	✓	✓	✓	✓	✓	✓	✓

Table 1: **Comparison with other popular simulators.** [A]Plasticine lab offers differentiable simulation, but doesn't provide gradient checkpointing, hence only supporting gradient flow over relatively short horizons. [B]FleX demonstrates the capability of smoke simulation in their demo video, but this feature and smoke rendering are not available in their released codebase.

There exist many simulation engines and environments for robotics research, and here we compare FluidLab and FluidEngine with some popular ones among them, including PyBullet (Coumans & Bai, 2016), MuJoCo (Todorov et al., 2012), FleX (Macklin et al., 2014), SoftGym (Lin et al., 2020), PlasticineLab (Huang et al., 2021), SAPIEN (Xiang et al., 2020), TDW (Gan et al., 2020) and DEDO (Antonova et al., 2021). Table 1 shows a comparison in terms of differentiability and supported material types. FluidLab covers a wider range of materials compared to existing environments. In addition, most of these environments rely on physics engines written in C++, and some are not open-sourced (such as FleX), making interpretability and extensibility a major hurdle, while FluidLab is developed in Python and Taichi, enjoying both GPU-acceleration and user-extensibility.

4 FLUIDLAB MANIPULATION TASKS

FluidLab presents a collection of complex fluid manipulation tasks with different fluid materials and their interactions with other objects. In general, we observe that manipulations tasks involving fluids can be categorized into two main groups: i) where fluid functions as a *transmitting media* and is used to manipulate other objects which are otherwise unreachable, and ii) where fluid is the *manipulation target* and the goal is to move or deform the fluid volume to a desired configuration. We present a suite of 10 different tasks in FluidLab, as shown in Figure 2, where in each task an agent needs to manipulate or interact with the fluids in the scene to reach a specific goal configuration, falling under the aforementioned two categories.

4.1 DETAILS OF EVALUATION TASKS

FluidLab contains 10 different manipulation tasks presented in Figure 2, all of which are motivated by real-world scenarios where an agent interacts with certain types of fluids.

Latte Art (Pouring & Stirring) We consider two types of latte art making tasks: the first one is making latte art via pouring, where an agent pours milk into a cup of coffee to form a certain pattern; the second one is via stirring, where a layer of viscous frothed milk covers a cup of coffee, and the agent needs to manipulate a rigid stick to stir in the cup to reach a specified goal pattern. Note that due to the nonlinear fluid dynamics governing the system, these two tasks are much more complex than simple drawing on a static canvas. For example, during pouring, where a milk particle ends up being at the end of the task may be far away from where it was originally poured from, due to continuous interaction with the coffee beneath it and motion of the fluid body caused by pouring.

Ice Cream (Static & Dynamic) This task involves policy learning concerning a non-Newtonian fluid – ice cream. We consider two settings here. The first one is a relatively simple and static setting, where we control an ice cream emitter that moves freely in space and emits ice cream into

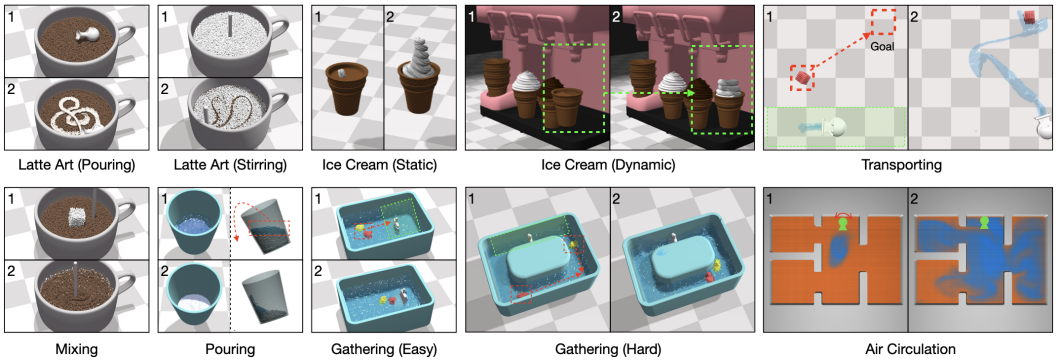

Figure 2: 10 Fluid manipulation tasks proposed in FluidLab. In each task, 1 depicts the initial condition and 2 shows a desired goal configuration. Both top and side views are shown for Pouring.

a cone and form a desired spiral shape. The second setting is more dynamic, where ice cream is squeezed out from a machine and an agent controls the biscuit cone. The goal configuration is defined with respect to the cone itself. During the process, the agent needs to reason about the dynamics of the icecream and its interaction with the cone since the ice cream moves together with the cone due to friction.

Transporting A water jet bot is required to move an object of interest to a randomly sampled goal location. The agent is only allowed to move within the green area, and thus can only transport the object using the water it injects.

Mixing Given a cube of sugar put in a cup of coffee, the agent needs to plan an sequence of stirring action and mixes the sugar particles with the coffee in the shortest period of time.

Fined Grained Pouring Alass filled with two types of non-Newtonian liquid with different densities, colored as blue and white respectively. The task is to pour the light blue liquid out of the glass while retaining as much white heavy liquid as possible.

Gathering (Easy & Hard) An agent equipped with a rigid plate is able to paddle the water in a tank, but its motion is restricted within the green area. The goal is to make water currents so that the two floating objects will be gathered towards the marked goal location. The easy setting has a relatively open space while in the hard setting, the spatial configuration is more complex and the two objects need to move in different directions.

Air Circulation The agent controls the orientation of an air conditioner (denoted in green in the figure) which generates cool air (colored in blue) in a house with multiple rooms, initially filled with warm air (colored in orange), and the goal is to cool down only the upper left and the right room, while ensuring the lower left room maintains at its original temperature. Completing this challenging task requires fine-grained control and reasoning over the motion of the cooling air.

4.2 Task and Action Representations

We model each task as an finite-horizon Markov Decision Process (MDP). An MDP consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, and a reward function associated with each transition step $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$. The transition function is determined by the physics simulation in FluidLab. We discuss the loss and reward for each task used for experiments in Section 5. The goal of the agent in the environment is to find a policy $\pi(a|s)$ that produces an action given the current state, and maximize the expected total return $E_\pi[\Sigma_{t=0}^{T} \gamma^t \mathcal{R}(s_t, a_t)]$, where $\gamma \in (0, 1)$ denotes the discount factor and T is the horizon of the environment.

State, Observation, and Action Space We represent MPM-based materials with a particle-based representation, using a $N_p \times d_p$ state matrix, where N_p is the number of all particles in the system, and $d_p = 6$ contains the position and velocity information of each particle. To represent the gas field, which is simulated using a grid-based representation, we use another $N_c \times d_c$ matrix to store information of all grid cells, where N_c is the number of cells in the grid, and d_c is a task-dependent vector. In the air circulation task, $d_c = 4$ contains the velocity (a 3-vector) and temperature (a scalar) of each cell. In addition, the state also includes the 6D pose and velocity (translational and rotational)

information of the end-effector of the robotic agent in the scene. All the tasks use an MPM grid of size 64^3, with around 100,000 particles, and a 128^3 grid for gas simulation. For RL algorithms, we down sample the number of particles and cells for each fluid body in the scene to avoid a intractably huge observation space. We do not evaluate image-based RL methods as many tasks FluidLab are 3D and occlusion could be a potential problem, and leave such evaluations for future work. The full action space of FluidLab is a 6-vector, containing both translational and angular velocity of the agent's end-effector. Each task has a task-specific action range and action space, with certain dimensions of the full action-space locked. More details of task settings are in Appendix C.

5 EXPERIMENTS

We evaluate model-free RL algorithms, sampling-based optimization methods as well as trajectory optimization using differentiable physics coupled with our proposed optimization techniques in FluidLab. We discuss loss and reward design, optimization techniques, and result findings in this Section.

5.1 LOSS AND REWARD

Latte Art (Pouring), **Latte Art (Stirring)**, **Ice Cream (Static)** and **Ice Cream (Dynamic)**: These tasks have a set of predefined goal patterns, and the loss is defined by summing up the Chamfer distance between particles in the current state of the scene and the particles in the goal pattern at each time step. Ideally, the goal pattern should be a static goal shape of the final step, but using a single goal pattern would be extremely challenging for RL methods: if only the last step of the trajectory is used for reward computation, such an extremely sparse reward will make learning infeasible; computing a dense reward based on the distance between the current state of each timestep and the goal pattern will also be non-informative: due to the dynamic movement within the fluid bodies, mimicing the static goal pattern at each timestep will not result in successful learning. Therefore, in order to make an informative comparison, we use a temporal trajectory of the desired goal pattern to compute the loss $L = \Sigma_t \mathrm{CD}(s_t, s_t^{goal})$, where CD denotes Chamfer Distance.

Transporting, **Pouring**, **Gathering (Easy)** and **Gathering (Hard)**: These tasks use a single spatial location as the goal and the loss is computed as: $\mathcal{L} = \Sigma_i \|p_i - p_{goal}\|$, which is summed over all particles of the object(s) to be manipulated. For **Pouring**, we apply this loss between the particles of the upper layer fluid and the ground location, with an additional loss attracting the particles of the lower layer fluid to their initial positions.

Mixing The goal of this task is to mix the sugar particles with the coffee. The loss is designed to maximize the spatial coverage of the sugar particles and is defined as $\mathcal{L} = -\Sigma_i \Sigma_j \|p_i - p_j\|$, where both i and j iterate over all particles originally belonging to the sugar cube.

Air circulation There are 27 sensors placed in the house, with 9 sensors in each room uniformly. The loss is defined as $\mathcal{L} = -\Sigma_{\mathrm{ul}}|T_{\mathrm{ul}} - T_{\mathrm{cool}}| + \Sigma_{\mathrm{r}}|T_{\mathrm{r}} - T_{cool}| + \Sigma_{\mathrm{ll}}|T_{\mathrm{ll}} - T_{\mathrm{warm}}|$, where T represents temperature, and ul, r and ll denote sensors in the upper left, right and lower left room respectively.

We use the losses discussed above for optimization-based methods; for model-free RL methods, the reward of each task is simply defined as $\mathcal{R} = c_1 - c_2\mathcal{L}$, where c_1 and c_2 are constants to ensure a reasonable reward scaling for each task.

5.2 TECHNIQUES AND OPTIMIZATION SCHEMES USING DIFFERENTIABLE PHYSICS

Differentiable simulation provides useful gradient information that can be leveraged for policy optimization. However, oftentimes the dynamics function could both be highly non-smooth (e.g. in case of a contact collision, where the gradient is hard to obtain), and exhibit a complex optimization landscape where the optimizer can easily get trapped in local minima. We discuss a few techniques and optimization schemes used in our experiments here for obtaining more informative gradients and leveraging them more effectively in the context of fluid manipulation.

Soft Contact Model In our simulation, the contact between particle-based fluid bodies and SDF-based rigid objects occurs in the grid operations. Such contact induces drastic state changes during the classical MPM process. We adopt a softened contact model in our simulation: we consider

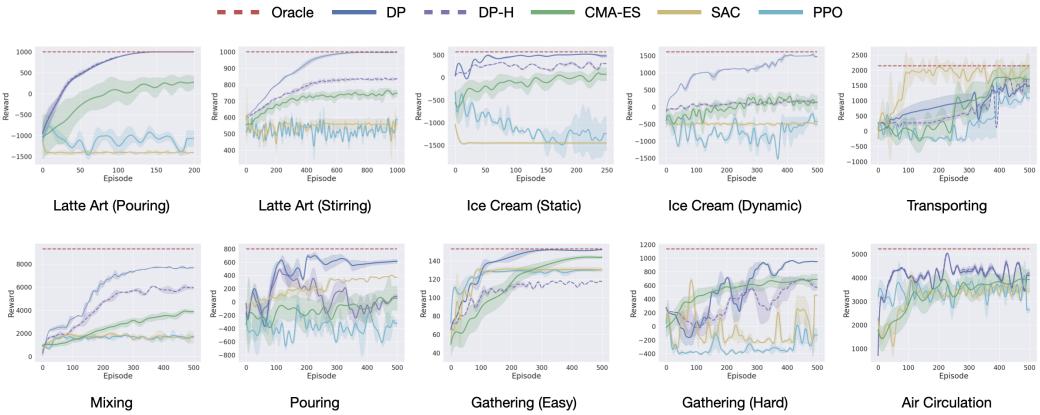

Figure 3: Reward learning curve for all evaluated methods in each task.

Task	Latte Art (Pour.)	LatteArt (Stir.)	Ice Cream (Sta.)	Ice Cream (Dyn.)	Transporting
SAC	-1417.6 $\pm$ 37.4	555.4 $\pm$ 22.8	-1450.7 $\pm$ 14.2	-509.6 $\pm$ 30.9	**2050.6 $\pm$ 450.7**
PPO	-1078.3 $\pm$ 187.4	585.3 $\pm$ 21.7	-1250.6 $\pm$ 563.8	-424.0 $\pm$ 110.5	1104.6 $\pm$ 99.7
CMA-ES	279.9 $\pm$ 124.1	745.2 $\pm$ 22.1	69.5 $\pm$ 140.1	131.4 $\pm$ 100.5	1694.5 $\pm$ 463.6
DP-H	**998.7 $\pm$ 0.5**	830.1 $\pm$ 7.2	295.1 $\pm$ 14.2	-162.9 $\pm$ 30.5	1443.9 $\pm$ 78.6
DP	**999.3 $\pm$ 0.7**	**995.4.6 $\pm$ 1.68**	**474.6 $\pm$ 21.4**	**1469.4 $\pm$ 28.8**	1471.1 $\pm$ 127.7

Task	Mixing	Pouring	Gathering (Easy)	Gathering (Hard)	Air Circulation
SAC	1523.2 $\pm$ 389.4	374.5 $\pm$ 7.3	130.2 $\pm$ 2.4	477.2 $\pm$ 9.7	4430.2 $\pm$ 264.7
PPO	1728.3 $\pm$ 159.4	-324.3 $\pm$ 50.3	129.7 $\pm$ 1.2	-138.9 $\pm$ 67.3	4282.2 $\pm$ 610.9
CMA-ES	3875.2 $\pm$ 143.2	57.1 $\pm$ 169.7	143.1 $\pm$ 1.4	687.2 $\pm$ 80.1	3917.3 $\pm$ 385.3
DP-H	5974.0 $\pm$ 198.8	88.3 $\pm$ 9.1	116.9 $\pm$ 0.3	566.9 $\pm$ 30.4	**5046.3 $\pm$ 23.7**
DP	**7648.4 $\pm$ 47.0**	**603.4 $\pm$ 29.7**	**151.5 $\pm$ 0.3**	**946.8 $\pm$ 5.1**	**5043.3 $\pm$ 21.2**

Table 2: The final accumulated reward and the standard deviation for each evaluated method.

contact occurs when the particles and the rigid objects are within a certain distance threshold ϵ, and compute an after-collision velocity v_c. Afterwards, we blend this velocity with the original velocity of the objects using a distance-based blending factor α, given by $v_{\text{new}} = \alpha v_c + (1-\alpha)v_{\text{original}}$, where $\alpha = \min\{\exp(-d), 1)\}$ and d is the distance between the two objects. This provides a smoothed contact zone and thus offers smoothed gradient information during contact.

Temporally Expanding Optimization Region During gradient-based trajectory optimization, if we consider the full loss computed using the whole trajectory with respect to a certain goal configuration, the whole optimization landscape could be highly nonconvex and results in unstable optimization. In our experiments, we warm start our optimization with a restricted temporal region of the trajectory and keeps expanding this region steadily whenever a stabilized optimization spot is reached for the current region. This greatly helps stabilize the optimization process using differentiable physics and produces better empirical results, as we show in Section 5.3.

Gradient Sharing within Fluid Bodies Many optimization scenarios involving fluids could be highly non-convex. Consider a simple pouring task where the goal is to pour all the water from a mug, with the loss defined as the distance between the ground floor and all the water particles. During the exploration phase, if a water particle was never poured out of the mug, it is trapped in a local minimum and the loss gradient won't be able to guide towards a successful pouring. Thanks to the continuity of fluid materials, we are able to share gradients between adjacent particles in a fluid body and use those informative gradients to guide other particles. Intuitively, once some *lucky* particles reach a better optimization spot during action sampling, (e.g. when it flows out of the mug), it gains useful gradients and can pass this information to neighboring particles via gradient sharing. In our experiments where fluid is required to reach a goal configuration, we enable gradient sharing by updating the gradient on each particle via accumulating neighboring particle gradients as follows: $\frac{\partial \mathcal{L}}{\partial q_i} = \Sigma_j w(\mathcal{L}_j, d_{ij}) \frac{\partial \mathcal{L}}{\partial q_j}$, where j iterates over all neighboring particles of i, q represents particle properties such as velocity or position, and w is a kernel function conditioned on both the distance between i and j and the loss of particle j, assigning a higher weight to neighboring particles who incurred a lower loss at the previous iteration.

5.3 METHOD EVALUATION WITH FLUIDLAB TASKS

We evaluate our proposed optimization schemes coupled with differentiable physics (DP), model-free RL algorithms including Soft Actor-Critic (SAC) (Haarnoja et al., 2018) and Proximal Policy Optimization (PPO) (Schulman et al., 2017), as well as CMA-ES (Hansen & Ostermeier, 2001), a sampling-based trajectory optimization method. We use the same accumulated reward to evaluate all the methods. Additionally, we include the performance of an oracle policy controlled by a human operator as a reference. The reward learning curves of the evaluated methods in all tasks are shown in Figure 3, and we report the final performance of all methods upon convergence in Table 2. Readers are encouraged to refer to our project page for more qualitative results.

Model-free RL Algorithms RL algorithms demonstrate reasonable learning behaviors in tasks with relatively small observation spaces, such as Transporting (where only a subset of water particles are present in the scene at each step) and Air Circulation. However, in environments such as Latte Art, Mixing, and Ice Cream, RL methods struggle to generate a reasonable policy. We suspect that this is because 1) the high-dimensional space used to describe the fluid system, which is hard for RL algorithms to abstract away useful information about the system, and 2) the non-linear dynamics of the complex fluids such as viscous frothed milk and the non-Newtonian ice creams, as well as their interactions with other solid materials, make policy search of RL policies based on sampled actions extremely difficult since randomly sampled actions and rewards are not informative enough in conveying contact and deformation information about the fluid systems.

Sampling-based trajectory optimization CMA-ES does reasonably well in many tasks with simple spatial configuration as goals. However, it struggles in fine-grained shape matching tasks (*e.g.* Ice Cream) and fine-grained control tasks (*e.g.* pouring). This is because CMA-ES searches for action sequences via trajectory sampling, and thus cannot handle delicate interactions with fluid materials well.

Differentiable physics-based trajectory optimization In most of the tasks, the gradients provided by our differentiable simulation are able to guide trajectory optimization towards a good optimization spot, resulting in near-oracle performance. It also demonstrates better sample-efficiency than other methods in most tasks. We additionally compare with a *hard* version of differentiable physics (DP-H), which uses the gradient to optimize trajectory, but without using the optimization techniques proposed in Section 5.2. DP-H does similarly well on tasks with relatively simple settings, but fails to produce good trajectories in scenarios with interactions between fluids and other materials bodies, as well as in tasks with more complex and non-convex optimization landscapes, such as Pouring with fine-grained control objective. The experimental results demonstrate that the gradient information provided by our differentiable simulator, coupled with the optimization schemes we proposed, provides informative optimization information when dealing with complex fluid systems.

6 CONCLUSION AND FUTURE WORK

We presented FluidLab, a virtual environment that covers a versatile collection of complex fluid manipulation tasks inspired by real-world scenarios. We also introduced FluidEngine, a backend physics engine that can support simulating a wide range of materials with distinct properties, and also being fully differentiable. The proposed tasks enabled us to study manipulation problems with fluids comprehensively, and shed light on challenges of existing methods, and the potential of utilizing differentiable physics as an effective tool for trajectory optimization in the context of complex fluid manipulation.

One future direction is to extend towards more realistic problem setups using vision as input. One potential way is to distill policy optimized using differentiable physics into neural-network based policies (Lin et al., 2022a), or use gradients provided by differentiable simulation to guide policy learning (Xu et al., 2022). We also plan to test policies trained using our simulation in real-world setups. Furthermore, since MPM was originally proposed, researchers have been using it to simulate various realistic materials with intricate properties, such as snow, sand, mud, and granular materials. These more sophisticated materials can be seamless integrated into the MPM simulation pipeline in FluidLab, and will be helpful for studying manipulation tasks dealing with more diverse set of materials. We believe the standardized task suites and the general purpose simulation capability of FluidLab will benefit future research in robotic manipulation involving fluids and other materials.

REFERENCES

Rika Antonova, Peiyang Shi, Hang Yin, Zehang Weng, and Danica Kragic Jensfelt. Dynamic environments with deformable objects. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. 5

Rika Antonova, Jingyun Yang, Krishna Murthy Jatavallabhula, and Jeannette Bohg. Rethinking optimization with differentiable simulation from a global perspective. *arXiv preprint arXiv:2207.00167*, 2022. 2, 3

Robert Bridson. *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015. 1

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 2, 3

Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pp. 297–307. PMLR, 2022. 3

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016. 3, 5

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018. 3

Chuang Gan, Jeremy Schwartz, Seth Alter, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, Megumi Sano, et al. Threedworld: A platform for interactive multi-modal physical simulation. *arXiv preprint arXiv:2007.04954*, 2020. 2, 3, 5

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018. 9

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001. 9

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018. 4

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019a. 3

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):1–16, 2019b. 2, 3

Zhiao Huang, Yuanming Hu, Tao Du, Siyuan Zhou, Hao Su, Joshua B Tenenbaum, and Chuang Gan. Plasticinelab: A soft-body manipulation benchmark with differentiable physics. *arXiv preprint arXiv:2104.03311*, 2021. 3, 4, 5

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017. 2, 3

Sizhe Li, Zhiao Huang, Tao Du, Hao Su, Joshua B Tenenbaum, and Chuang Gan. Contact points discovery for soft-body manipulations with differentiable physics. *arXiv preprint arXiv:2205.02835*, 2022. 2, 3

Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B Tenenbaum, and Antonio Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*, 2018. 3

Junbang Liang, Ming Lin, and Vladlen Koltun. Differentiable cloth simulation for inverse problems. *Advances in Neural Information Processing Systems*, 32, 2019. 3

Xingyu Lin, Yufei Wang, Jake Olkin, and David Held. Softgym: Benchmarking deep reinforcement learning for deformable object manipulation. *arXiv preprint arXiv:2011.07215*, 2020. 1, 2, 3, 5

Xingyu Lin, Zhiao Huang, Yunzhu Li, Joshua B Tenenbaum, David Held, and Chuang Gan. Diffskill: Skill abstraction from differentiable physics for deformable object manipulations with tools. *arXiv preprint arXiv:2203.17275*, 2022a. 2, 3, 9

Xingyu Lin, Yufei Wang, Zixuan Huang, and David Held. Learning visible connectivity dynamics for cloth smoothing. In *Conference on Robot Learning*, pp. 256–266. PMLR, 2022b. 3

Pingchuan Ma, Yunsheng Tian, Zherong Pan, Bo Ren, and Dinesh Manocha. Fluid directed rigid body control using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 37(4): 1–11, 2018. 3

Miles Macklin. Warp: A high-performance python framework for gpu simulation and graphics. https://github.com/nvidia/warp, March 2022. NVIDIA GPU Technology Conference (GTC). 3

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014. 2, 3, 5

Gautham Narasimhan, Kai Zhang, Ben Eisner, Xingyu Lin, and David Held. Self-supervised transparent liquid segmentation for robotic pouring. In *2022 International Conference on Robotics and Automation (ICRA)*, pp. 4555–4561. IEEE, 2022. 3

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020. 3

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pp. 8459–8468. PMLR, 2020. 3

Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, pp. 317–335. PMLR, 2018. 1, 2, 3

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 9

Daniel Seita, Yufei Wang, Sarthak Shetty, Yao Li, Zackory Erickson, and David Held. Toolflownet: Robotic manipulation with tools via predicting tool flow from point clouds. In *Conference on Robot Learning*. PMLR, 2022. 1, 3

Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain. Time-contrastive networks: Self-supervised learning from video. In *2018 IEEE international conference on robotics and automation (ICRA)*, pp. 1134–1141. IEEE, 2018. 1

Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 121–128, 1999. 4

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)*, 32(4):1–10, 2013. 4

Francisco Suárez-Ruiz, Xian Zhou, and Quang-Cuong Pham. Can robots assemble an ikea chair? *Science Robotics*, 3(17):eaat6385, 2018. 3

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012. 3, 5

Hsiao-Yu Fish Tung, Zhou Xian, Mihir Prabhudesai, Shamit Lal, and Katerina Fragkiadaki. 3d-oes: Viewpoint-invariant object-factorized environment simulators. *arXiv preprint arXiv:2011.06464*, 2020. 3

Yilin Wu, Wilson Yan, Thanard Kurutach, Lerrel Pinto, and Pieter Abbeel. Learning to manipulate deformable objects without demonstrations. *arXiv preprint arXiv:1910.13439*, 2019. 3

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. Sapien: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11097–11107, 2020. 2, 3, 5

Jie Xu, Tao Chen, Lara Zlokapa, Michael Foshey, Wojciech Matusik, Shinjiro Sueda, and Pulkit Agrawal. An end-to-end differentiable framework for contact-aware robot design. *arXiv preprint arXiv:2107.07501*, 2021. 2, 3

Jie Xu, Viktor Makoviychuk, Yashraj Narang, Fabio Ramos, Wojciech Matusik, Animesh Garg, and Miles Macklin. Accelerated policy learning with parallel differentiable simulation. *arXiv preprint arXiv:2204.07137*, 2022. 2, 3, 9

Ben Yang, Youquan Liu, Lihua You, and Xiaogang Jin. A unified smoke control method based on signed distance field. *Computers & graphics*, 37(7):775–786, 2013. 3

Andy Zeng, Pete Florence, Jonathan Tompson, Stefan Welker, Jonathan Chien, Maria Attarian, Travis Armstrong, Ivan Krasin, Dan Duong, Vikas Sindhwani, et al. Transporter networks: Rearranging the visual world for robotic manipulation. *arXiv preprint arXiv:2010.14406*, 2020. 3

Bo Zhu, Michiaki Iwata, Ryo Haraguchi, Takashi Ashihara, Nobuyuki Umetani, Takeo Igarashi, and Kazuo Nakazawa. Sketch-based dynamic illustration of fluid systems. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, pp. 1–8, 2011. 3

A SAMPLE CODE FOR BUILDING A SIMULATION ENVIRONMENT USING FLUIDENGINE

```
import os
import gym
import numpy as np
from gym.spaces import Box
from fluidlab.configs.macros import *
from fluidlab.engine.scenes import Scene
import fluidlab.utils.misc as misc_utils

class FluidEnv(gym.Env):
    '''
    Base env class.
    '''
    def __init__(self, version, loss=True, loss_type='diff', seed=None):
        if seed is not None:
            self.seed(seed)

        self.horizon               = 500
        self.horizon_action        = 500
        self.target_file           = None
        self._n_obs_ptcls_per_body = 200
        self.loss                  = loss
        self.loss_type             = loss_type
        self.action_range          = np.array([-1.0, 1.0])

        # create a simulation
        self.scene = Scene()
        self.build_env()
        self.gym_misc()

    def seed(self, seed):
        super(FluidEnv, self).seed(seed)
        misc_utils.set_random_seed(seed)

    def build_env(self):
        self.setup_agent()
        self.setup_statics()
        self.setup_bodies()
        self.setup_gas_field()
        self.setup_boundary()
        self.setup_renderer()
        self.setup_loss()

        self.scene.build()
        self._init_state = self.scene.get_state()

        print(f'===>  {type(self).__name__} built successfully.')

    def setup_agent(self):
        agent_cfg = CfgNode(new_allowed=True)
        agent_cfg.merge_from_file(get_cfg_path('agent_scooping.yaml'))
        self.scene.setup_agent(agent_cfg)
        self.agent = self.scene.agent

    def setup_statics(self):
        # add static mesh-based objects in the scene
        self.scene.add_static(
            file='tank.obj',
            pos=(0.5, 0.4, 0.5),
            euler=(0.0, 0.0, 0.0),
            scale=(0.85, 0.92, 0.92),
            material=TANK,
```

```
            has_dynamics=False,
        )

    def setup_bodies(self):
        # add fluid/object bodies
        self.scene.add_body(
            type='cube',
            lower=(0.12, 0.25, 0.18),
            upper=(0.88, 0.45, 0.82),
            material=WATER,
        )
        self.scene.add_body(
            type='mesh',
            file='duck.obj',
            pos=(0.5, 0.6, 0.55),
            scale=(0.10, 0.10, 0.10),
            euler=(0, -90.0, 0.0),
            color=(1.0, 1.0, 0.3, 1.0),
            filling='grid',
            material=ELASTIC,
        )
        self.scene.add_body(
            type='cylinder',
            center=(0.2, 0.5, 0.55),
            height=0.08,
            radius=0.02,
            euler=(0.0, 0.0, -45.0),
            color=(1.0, 0.5, 0.5, 1.0),
            filling='natural',
            material=ELASTIC,
        )

    def setup_gas_field(self):
        self.scene.setup_gas_field(
            res          = 128,
            dt           = 0.03,
            solver_iters = 50,
            decay        = 0.99,
            q_dim        = 1,
        )

    def setup_boundary(self):
        self.scene.setup_boundary(
            type='cube',
            lower=(0.12, 0.25, 0.18),
            upper=(0.88, 0.95, 0.82),
        )

    def setup_renderer(self):
        self.scene.setup_renderer()

    def setup_loss(self):
        pass

    def gym_misc(self):
        obs = self.reset()
        self.observation_space = Box(DTYPE_NP(-np.inf), DTYPE_NP(np.inf),
 obs.shape, dtype=DTYPE_NP)
        if self.scene.agent is not None:
            self.action_space = Box(DTYPE_NP(self.action_range[0]),
DTYPE_NP(self.action_range[1]), (self.scene.agent.action_dim,), dtype
=DTYPE_NP)
        else:
            self.action_space = None

```

```
    def reset(self):
        self.scene.set_state(**self._init_state)
        return self._get_obs()

    def step(self, action):
        self.scene.step(action)

        obs    = self._get_obs()
        reward = self._get_reward()

        assert self.t <= self.horizon
        if self.t == self.horizon:
            done = True
        else:
            done = False

        info = dict()
        return obs, reward, done, info

    def render(self, mode='human'):
        assert mode in ['human', 'rgb_array']
        return self.scene.render(mode)
```

B Details on FluidLab's Differentibility and Rendering

Differentiability and gradient checkpointing We use the automatic differentiation tool (autodiff) provided by Taichi to get gradients for most of our computations; for more complex computations and kernels not supported by Taichi's autodiff, we implemeneted gradient computation ourselves, including SVD computations and the projection step in gas simulation. One additional challenge for making the whole simulation fully differentiable is enabling gradient flow through time, which requires bookkeeping system states of all timesteps. Since most of FluidLab's tasks have tens of thousands of simulation steps, it is impossible to fit the whole computation graph into the GPU memory. In order to address this, we implemented efficient gradient checkpointing to enable gradient flow over unlimited time horizons. During the forward process of the simulation, only a local trajectory of the environment states are stored in the GPU memory for fast computation. Whenever the GPU memory is full, we store the latest state of the environment to a caching device (either CPU memory or local disks), and clear up the GPU memory for further simulation. Once a complete forward pass is done, we restore the cached states and run forward simulation again to fulfill the GPU memory with the local chunk of simulation trajectory. Note that instead of caching all intermediate states, we only store one single step of the states at every caching step, and run simulation again to restore the states of the local trajectory; therefore, the i/o step of such caching operations is very efficient. This enables us to propagate loss and gradient information backward in time through a simulation trajectory with arbitrary number of steps, unconstrained by the GPU memory size.

Rendering We implemented a photo-realistic renderer for visualizing scenes created with FluidEngine. The renderer is developed in C++ using OpenGL, supports GPU-accelerated real-time rendering. Part of our rendering pipeline was modified based on FleX's rendering system, enhanced with various features such as particle-level colorization, headless rendering, dynamic object loading, smoke rendering, etc. We also provided a set of python APIs to dynamically update the scene configuration from within Python. The renderer supports rendering materials as either particles or fluids. Particle-based rendering is more interpretable for visualization and debugging purposes, while fluid-based rendering is more photo-realistic and allows possible image-based sim-to-real transfer applications in the future. We present a comparison of these two rendering methods in Figure 4.

C FluidLab Task Details

We present our task-specific settings for action spaces, observed number of particles per fluid body and observed number of grid cells in Table 3. Detailed properties of all materials used are listed in Table 4.

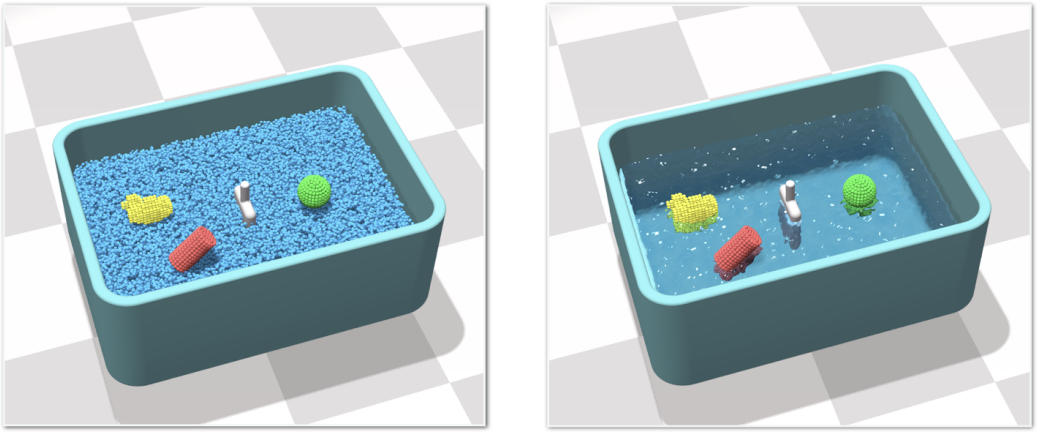

Figure 4: A rendered scene with water and floating objects. Left: water rendered as particles. Right: realistic water rendering.

Task	Action space		number of particles	number of grid cells
	Trans. Velocity	Rot. Velocity	per fluid body	for gas field
Latte Art (Pouring)	Along x and z	-	1000	-
Latte Art (Stirring)	Along x and z	-	1000	-
Ice Cream (Static)	Along x, y and z	-	2000	-
Ice Cream (Dynamic)	Along x, y and z	-	2000	-
Transporting	Along x and z	Around y	500	-
Mixing	Along x and z	-	1000	-
Pouring	-	Around z	500	-
Gathering (Easy)	Along x and z	-	500	-
Gathering (Hard)	Along x and z	-	500	-
Air Circulation	-	Around y	-	$12 \times 12 \times 12$

Table 3: Task-specfic settings.

Material	Type	Lamé parameters		Density ρ
		μ	λ	
Water	Liquid	0.0	277.78	1.0
Milk	Liquid	0.0	277.78	1.0
Frothed Milk	Viscous Liquid	208.33	277.78	1.0
Coffee	Liquid	0.0	277.78	1.0
Ice Cream	Non-newtonian Liquid	416.67	277.78	0.5
Floating object	Elastic	416.67	277.78	0.5
Sugar cube	Viscous Liquid	208.33	277.78	1.0
Light Liquid in Pouring	Liquid	0.0	277.78	0.8
Heavy Liquid in Pouring	Liquid	0.0	277.78	1.5
Object in Transporting	Rigid	416.67	277.78	5.0

Table 4: Details of materials properties used in FluidLab.