Loki: A Unified Multiphysics Simulation Framework for Production

STEVE LESSER, ALEXEY STOMAKHIN, GILLES DAVIET*, JOEL WRETBORN, JOHN EDHOLM, NOH-HOON LEE, ESTON SCHWEICKART, XIAO ZHAI, SEAN FLYNN, and ANDREW MOFFAT,
Weta Digital, New Zealand

We introduce Loki, a new framework for robust simulation of fluid, rigid, and deformable objects with non-compromising fidelity on any single element, and capabilities for coupling and representation transitions across multiple elements. Loki adapts multiple best-in-class solvers into a unified framework driven by a declarative state machine where users declare 'what' is simulated but not 'when,' so an automatic scheduling system takes care of mixing any combination of objects. This leads to intuitive setups for coupled simulations such as hair in the wind or objects transitioning from one representation to another, for example bulk water FLIP particles to SPH spray particles to volumetric mist. We also provide a consistent treatment for components used in several domains, such as unified collision and attachment constraints across 1D, 2D, 3D deforming and rigid objects. Distribution over MPI, custom linear equation solvers, and aggressive application of sparse techniques keep performance within production requirements. We demonstrate a variety of solvers within the framework and their interactions, including FLIP-style liquids, spatially adaptive volumetric fluids, SPH, MPM, and mesh-based solids, including but not limited to discrete elastic rods, elastons, and FEM with state-of-the-art constitutive models. Our framework has proven powerful and intuitive enough for voluntary artist adoption and has delivered creature and FX simulations for multiple major movie productions in the preceding four years.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Simulation types and techniques.*

Additional Key Words and Phrases: unified physics, coupling, movie production, distributed simulation

ACM Reference Format:
Steve Lesser, Alexey Stomakhin, Gilles Daviet, Joel Wretborn, John Edholm, Noh-hoon Lee, Eston Schweickart, Xiao Zhai, Sean Flynn, and Andrew Moffat. 2022. Loki: A Unified Multiphysics Simulation Framework for Production. *ACM Trans. Graph.* 41, 4, Article 50 (July 2022), 20 pages. https://doi.org/10.1145/3528223.3530058

1 INTRODUCTION

In the last few decades computer graphics researchers have developed a vast variety of techniques to simulate different kinds of physical phenomena. The focus has largely been on tuning each of these techniques to the specific needs of particular solvers, resulting in efficient specialized data structures, discretizations, integration schemes, and other distinct components of a solver. Coupling has always been a challenging task, as bringing together different and potentially incompatible solvers is non-trivial. Attempts to achieve coupling typically result in compromises which limit quality for individual elements compared to specialized solvers, or target strong coupling for a few selected phenomena and are not easily extensible.

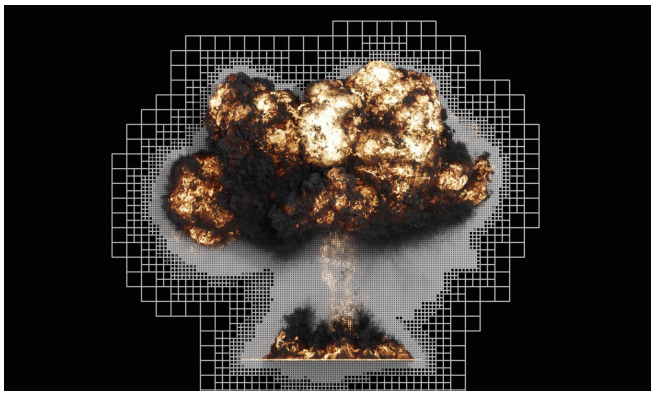


Fig. 1. **Explosion.** An explosion simulation performed using our adaptive Eulerian fluid solver. The image also shows the underlying Bucket topology: finer Buckets in the vicinity of the explosion, with gradual coarsening away from it to ensure enough world-space padding for capturing the surrounding air flow. Each Bucket represents an $8 \times 8 \times 8$ block of voxels for material channels, and a $4 \times 4 \times 4$ block of voxels for velocity and pressure channels. ©Wētā FX.

In order for a solver to efficiently 'talk' to any other solver, they each must adopt a set of shared rules in regard to integration, discretization, and scheduling. Defining and enforcing those shared rules can lead to a loss in performance or flexibility of the individual components if chosen too strictly, or they can lead to limited ability to interact if defined too loosely.

We present Loki, the first system targeting visual effects production to codify those rules, implement a large collection of solvers within those rules, and achieve a level and flexibility of interaction between different simulated physical systems that has not been demonstrated before. Importantly, Loki avoids the combinatorial complexity explosion of both backend accurate coupling and frontend configuration previously seen from mixing many solvers together, making coupling tractable for production. To manage performance, we include support for distributed simulations all the way down to the core data structures, and introduce several new specialized linear equation solvers for different types of linear systems commonly encountered in visual effects. This paper includes descriptions of the various algorithms and unifying components Loki uses, with emphasis on the overall architecture.

*Gilles Daviet is currently affiliated with NVIDIA

All images ©Wētā FX.

Authors' address: Steve Lesser, slesser@wetafx.co.nz, Alexey Stomakhin, st.alexey@gmail.com, Gilles Daviet, gdaviet@nvidia.com, Joel Wretborn, jwretborn@wetafx.co.nz, John Edholm, jedholm@wetafx.co.nz, Noh-hoon Lee, nlee@wetafx.co.nz, Eston Schweickart, eschweickart@wetafx.co.nz, Xiao Zhai, xzhai@wetafx.co.nz, Sean Flynn, sflynn@wetafx.co.nz, Andrew Moffat, amoffat@wetafx.co.nz, Weta Digital, New Zealand.

0730-0301/2022/7-ART50 $15.00
https://doi.org/10.1145/3528223.3530058

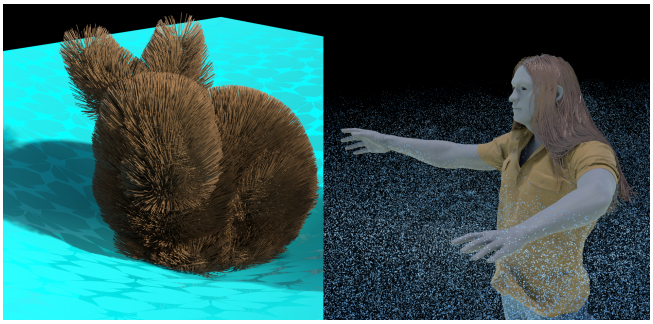


Fig. 2. **Interacting solids and swimming character.** Coupling between 1D, 2D, and 3D elastic solids (left) and a diver scene that demonstrates the simulation of free-surface fluids, elastic solids, and the coupling in between (right). Both simulations use the same user-facing Solver Setup. ©Wētā FX.

After reviewing related works in Section 2, we cover high-level goals, constraints, and resulting design principles in Section 3. We examine the user-facing architecture in Section 4, the developer-facing architecture in Section 5, and the shared solver components in Section 6. Several simulation case studies are covered in Section 7 to demonstrate flexibility of the system. Finally, we conclude with a discussion of strengths, limitations, and future work in Section 8.

2 RELATED WORKS

We limit our review to prior work targeting multiple phenomena for physics-based animation rather than systems targeting a single use case. We believe this highlights how a new architecture is needed to allow both best-in-class individual solvers and combinations of solvers that coexist in a single system to achieve production goals. We refer to methods using a single representation for simulations in multiple domains as multiphysics methods, and we refer to systems combining multiple representations for simulations as multiphysics frameworks.

2.1 Multiphysics methods

Position based dynamics [Macklin et al. 2016; Müller et al. 2007] is a popular choice for production soft body simulation as seen in Maya's Nucleus [Stam 2009], NVIDIA PhysX [Macklin et al. 2019], or Houdini's Vellum solvers. While fluids are supported [Macklin and Müller 2013; Yang et al. 2015], they are not as performant and scalable as Eulerian grid-based implementations [Fedkiw et al. 2001; Zhu and Bridson 2005]. Convergence is also strongly affected by discretization, slowing down for deep constraint hierarchies.

Elastons [Martin et al. 2010] offer a unified approach across dimensions for mesh-based objects, at the cost of a high overhead over reduced models for thin objects and fluids, and also the lack of an Eulerian representation.

Material point method [Jiang et al. 2016] offers excellent fidelity for some complex materials such as sand [Daviet and Bertails-Descoubes 2016; Klár et al. 2016] and snow [Stomakhin et al. 2013], phase transitions [Stomakhin et al. 2014], and has been extended to support simulation of codimensional elastic solids such as hair and cloth [Jiang et al. 2017]. However, grid resolution limitations for collisions and performance drawbacks make it difficult to compete with dedicated elastic FEM approaches or FLIP for inviscid water.

2.2 Multiphysics frameworks

Black box weak coupling is a technique for coupling multiple solvers by alternately isolating each solver and stepping it forward using the results from the other solvers as fixed inputs, e.g. [Akbay et al. 2018; Guendelman et al. 2005]. This allows some mutual interaction across the solvers in the result, but there is no reuse of shared components across the solvers; and given that the coupling scheme is rarely run to convergence, there are limitations for how accurate those interactions can be, in particular regarding momentum conservation. This concept is often extended to directional simulations in movie production, where one solver runs only after the earlier solver has completely finished; so there is only one-way interaction.

Monolithic approaches, such as [Brandt et al. 2019; Fei et al. 2018, 2017; Losasso et al. 2006; Lyu et al. 2021; Robinson-Mosher et al. 2008; Takahashi and Batty 2020, 2021; Teng et al. 2016] at the opposite end of the spectrum, propose a specialized solver targeting a particular set of phenomena. Their strength lies in strong convergence properties, but their lack of flexibility makes them unsuitable for the development of a comprehensive and extensible multiphysics framework. [Losasso et al. 2008; Patkar et al. 2013] explore transitions between different representation within a monolithic system.

Node graphs are a common approach to exposing an interface for developing solvers, using small building blocks representing execution kernels in a dependency; they are popular in visual effects software such as SideFX's Houdini as a way to package solvers. Their generality facilitates creativity and ease of experimentation for developers to try out new ideas and design new solvers. However the accompanying creative chaos can hinder productivity when one tries to assemble a solid production system, with well-established components that need to reliably interact with one another in any combination. For instance, there is essentially one way to assemble a FLIP solver, and one way to do strong coupling correctly. But none of that structure is enforced, and the 'proper' ways of doing multiphysics are easy to miss by technical artists and even experienced researchers amidst the ocean of possibilities those systems offer, especially as the number of phenomena increases.

Multiphysics C++ libraries such as *PhysBAM* [Dubey et al. 2011], *SOFA* [Faure et al. 2012], or *Chrono* [Tasora et al. 2016] contain solvers for simulating various phenomena such as deformable and rigid bodies, compressible and incompressible fluids, fracture, fire,

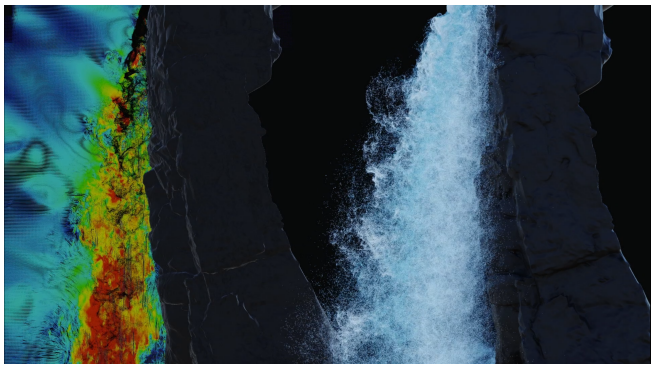


Fig. 3. **Waterfall.** A waterfall simulation demonstrating coupling between a FLIP fluid (right) and an Eulerian wind field (left). ©Wētā FX.

hair, cloth, muscles, and more. Shared data structures, generic interfaces, and powerful linear algebra routines allow for fast prototyping and design of new custom simulation drivers and integrators, which can be used for coupling multiple phenomena. However, the main target audience for such libraries are simulation researchers and engineers; and the complexities of setting up new multiphysics simulations make them out of reach for most VFX artists.

Domain specific languages (DSL), such as [Bernstein et al. 2016; Hu et al. 2019; Kjolstad et al. 2016], offer useful abstraction layers allowing developers to focus on physics and algorithms while avoiding dealing with specifics of parallel/distributed implementations on heterogeneous architectures.

3 THE ARCHITECTURE

We propose several goals and constraints for a simulation framework to be used by technical artists on a wide range of phenomena for movie production.

(1) *Quality.* Any single element including free-surface liquids, volumetric fluids, and rigid and deforming solids, can be simulated with the same quality as with a specialized solver. In particular, we aim to reproduce real-world results for use in photoreal visual effects.
(2) *Interactions.* Multiple elements can interact with each other in any combination including coupled, one-way coupled in any direction, and solved independently with no interactions.
(3) *Performance.* Simulations are interactive enough to set up on a single workstation, and performant enough to complete overnight on one or more machines for final quality photoreal results.
(4) *Usability.* Technical artists from junior to senior level, without deep knowledge of how to build solvers, can set up and complete simulations with enough controls to address creative feedback.
(5) *Reusable.* Features should be reusable wherever possible across solvers.
(6) *Hardware.* Machines running simulations will have many CPU cores, but may not have a GPU. Multiple machines may be available for large simulations.

There are currently no multiphysics methods that are able to handle all of the types of simulation that we require without a loss in quality. Furthermore, existing multiphysics frameworks lack fidelity, scale poorly with complexity, or lack the usability needed for artist tools. Therefore we have developed a new simulation framework called Loki to better meet both our backend and frontend requirements.

3.1 Design Principles

Our simulation framework's goals and constraints led us to define several design principles to stay true to the requirements in both the user facing controls and the backend architecture:

(1) *Best-in-Class.* If there is a high-fidelity technique widely shown to be best for a particular phenomenon we should favor including that technique in the framework rather than competing against it.
(2) *Configuration Scales with Phenomena.* The complexity of the user-facing configuration should scale linearly with the number of phenomena represented, not the number of objects being simulated.
(3) *Avoid Combinatorial Explosion.* When building a unified framework over many components, we should avoid needing to write a specialized coupling scheme between any two elements. For instance considering hair, solid, water, and gas, that would mean coupling hair with solids, hair with water, solids with water, hair with gas, solids with gas, water with gas, or more generally for n models, $n(n-1)/2$ coupling terms. Instead, we should devise abstractions that each model can follow to receive automatic coupling capabilities, keeping the implementation complexity to a minimum.
(4) *Sparse Parameters.* Parameters should have reasonable defaults and can be set sparsely at whatever granularity is required, such as constant across all objects, constant for a single object, or varying across vertices. If there is a conflict between parameters set for multiple granularities, then the more local parameter with the narrowest scope is used.
(5) *Physical by Default.* Default solver settings should prefer the most physically correct behavior available. As such, we should consistently use physical units, such as [kg/m^3] for volumetric mass, and validate against measured results. In a production context there are often cases where artistic vision should override physical correctness, so we should also provide non-physical tools to tweak the results of a simulation; but these options should be disabled by default.
(6) *Locality of Computation.* All algorithms should require only local neighborhood access instead of global access. This is in line with locality of differential operators used in continuum mechanics and the observation that the strength of many physical interactions decreases sharply with distance. This is particularly important for dividing work both locally and across multiple machines, where each unit of work may only have efficient access to a subset of the full simulation data. In cases where this is not possible, such as with long-range attachment between elements across the whole domain, or large rigid bodies inside a high-resolution fluid, we should provide overrides, at the cost of potentially increased communication overhead.

Although sometimes more aspirational than strict, these guidelines influence decisions around which solvers to incorporate, what parameter space to expose, and how to structure the user workflow into the complete system we call Loki. Our system integrates multiple traditional solvers into a unified distributed simulation framework driven by a declarative configuration proficient at high fidelity simulations involving single elements, coupling multiple elements, and transitioning between representations.

4 USER-FACING DESIGN

We will now examine the user-facing architectural elements of Loki, including the solver configuration and data flow, to understand how artists work with Loki. As we progress, we will define a number of Loki-specific terms; definitions for these are summarized in Section 10.

4.1 Declarative Configuration

The first major user-facing component of the architecture is the method of describing the behavior of a simulation, such as 'simulate curves as discrete elastic rods [Bergou et al. 2010, 2008] which are coupled via drag to an incompressible volumetric wind which exists in a narrow band around the curves' shown in Figure 4. Loki uses a declarative configuration called a Solver Setup to describe what should happen within the simulation consisting of a tree of nodes. There are two kinds of nodes: Behaviors and Groups. Behaviors are leaf-level nodes that describe a high-level unit of computation such as adding an object into the scene or operating on an existing object. Groups are intermediate-level nodes that are used for organization and scoping in the Solver Setup tree; they may contain Behaviors or other Groups. Figure 5 shows a Solver Setup used to describe hair in the wind which we will use as an example to walk through the general Solver Setup rules.

Groups collect related functionality to describe a high-level element in the scene, with the convention that each Group should describe a fully working solver for at least one element. If the Wind Group is disabled, there is still a fully functioning hair solver via the Hair Group and vice versa. Groups also provide scope for behaviors to control their influence on the current subtree; for example in Figure 5 the Pressure Material, enforcing incompressibility, applies to the Volume Object but not the Curves Object, while the Gravity Force applies to both the Curves Object and the Volume Object.

Behaviors may schedule multiple types of work during solver execution, such as the Volume Object handling emission, advection, and adaptivity criteria. Parameters on each behavior are used to control settings for how something is simulated; but in keeping with the Sparse Parameters principle, the default values are usually reasonable for physically based results, and they can be overridden at the behavior level or be fine tuned at the data level, such as with painted overrides. Following the Configuration Scales with Phenomena principle, Behaviors generally depend on data to flow in from the outside to define what is simulated, such as the Curves Object using an input data stream to describe which curves to simulate instead of building the curves directly by parameters on the Behavior. The exceptions are the Behaviors related to fluid particle emission, as they need to properly respect the notion of volumetric flux.

The Solver Setup does not require the user to describe any execution order across the Behaviors, since everything is scheduled

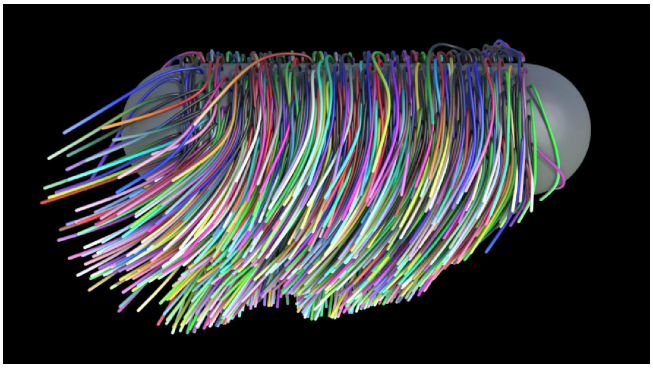


Fig. 4. An example scene for simulating hair and wind coupled through buoyancy and drag forces. ©Wētā FX.

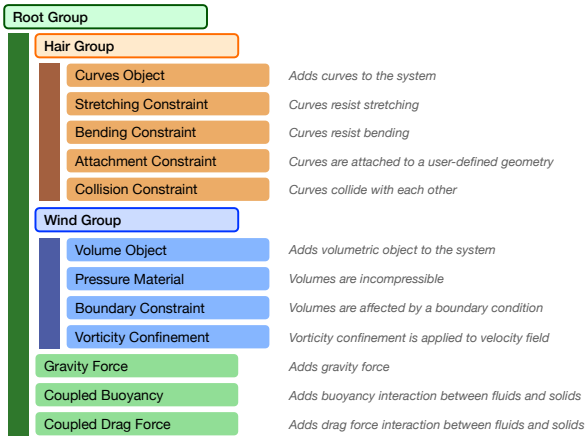


Fig. 5. The Solver Setup for the example shown in Figure 4. Root Group contains the Hair Group, the Wind Group, and Behaviors which span the two. Hair Group contains all the Behaviors needed to simulate the hairs as discrete elastic rods independent of anything else in the simulation. Wind Group contains all the Behaviors needed to simulate the wind as an incompressible fluid defined around the hair. Each Behavior affects all Objects within its current group and all of the child groups. ©Wētā FX.

automatically and is locked off from manipulation. This is an intentional trade-off of less flexibility in front-end scheduling, for simplicity and robustness of setups. Creating and managing node graphs of small kernels of execution describing simulation stages and dependencies more explicitly was considered, and would also have supported many of our requirements, but it places a much heavier burden on users to deeply understand how to build solvers and how to manage interactions for weak and strong coupling in all kinds of interactions. By moving to a higher level declarative model, we still provide the ability to configure any combination of objects and actions while removing the combinatorial complexity of defining their dependencies explicitly; and allow users to operate at a higher level than would be possible if the solvers were exposed at a low enough node graph level to meet the other requirements.

4.2 Data Flow

While the Solver Setup is declarative, we still need a way to manage the flow of data into and out of the solver. Regarding data preparation before the solver and post-processing afterwards, we have struggled to compete with major digital content creation (DCC) products the artists are already comfortable working within; so in keeping with the Best-in-Class principle, we embrace them and ensure Loki is available within multiple major applications. Loki provides plugins for these DCCs and heavily leans on them for data flow, in order to keep development efforts focused on the simulations. We build the Loki solvers, Solver Setup configuration, and a small node graph for feeding data into the solver setups all independently of any single DCC.

The flexibility gained in separating Loki data types and solvers from any single DCC has made it feasible to create and maintain Loki integration for seven separate applications, with the most usage seen in the SideFX Houdini plugin and Autodesk Maya plugin. This separation also gave Loki developers freedom to use custom data structures outside of those provided by the host applications,

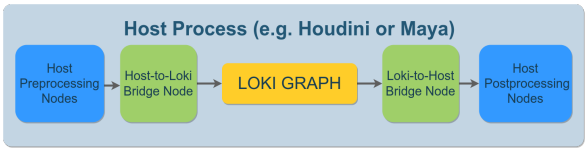


Fig. 6. Data flow between Loki and the host application is via a live bridge (represented as a plugin in the respective host), which handles the conversion between host and Loki data types. ©Wētā FX.

such as a spatially adaptive volume which natively supports distribution across multiple machines. We generalize the processes Loki runs within as host applications since several of them are not complete DCCs, such as a small stand-alone tool called wtLoki used for headless evaluation or debugging. To keep a consistent experience within these applications a common data flow pattern is followed, as seen in Figure 6:

(1) Native host application nodes are used to load and prepare data for simulation with the host application's node graph.
(2) A Host-to-Loki plugin node is used for converting host application data to Loki formats.
(3) A Loki Graph plugin node is used for executing the Loki node graph. This accepts the output of Host-to-Loki nodes or other Loki Graph nodes, and injects them into the Loki node graph. The Solver Setup exists as a special node within the Loki node graph for declarative configuration and execution of the solver itself.
(4) A Loki-to-Host plugin node is used for bridging Loki data back to the host application.
(5) Native host application nodes are used to do any post-processing, visualization, or publishing of the simulation results.

While seemingly wasteful to convert data between host application and Loki data types so often, we find in the large majority of our use cases the cost of conversion is small compared the total simulation runtime. The separation provides almost complete insulation for our data types, keeping most of Loki agnostic of any target application and the host application plugins narrow in scope, mostly focusing on data type conversions. We found this to be an acceptable cost for the high increase in productivity from artists using native tooling; and the strategy has led to major wins in both the Usability and Reusability goals.

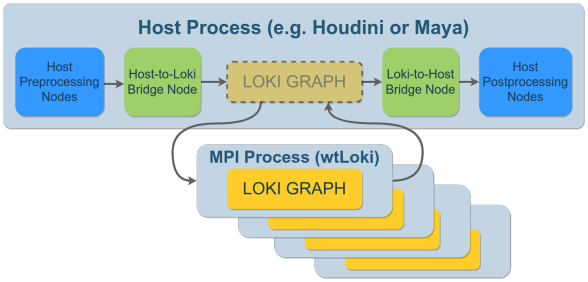


Fig. 7. In a distributed configuration, the host application process handles data processing upstream and downstream of the solver, while a group of separate wtLoki processes work together over MPI to partition the solver work and evaluate the Loki graph containing the Solver Setup. ©Wētā FX.

We provide a custom UI for configuring the Solver Setup and building the node graph that connects data between the host applications and the Solver Setup. This UI also provides functionality for nominating important properties to be directly controllable in the host application by 'exposing' them to the host application so users can manipulate them with native controls without needing to open the Loki UI. Exposing properties allows senior artists to build templates with a Loki configuration they would like to reuse as native-looking nodes in the host application, and pass them on to other artists who may not need to open the Loki UI, or even know they are using Loki. Loki integrates fine-grained control of individual simulation elements into artist workflows through the preservation of attribute channels on top of incoming host application data, which the Solver Setup then uses to override solver settings at a per-object or per-primitive level. This lets artists spatially vary and animate the material properties of the simulation components, utilizing standard tooling.

When running distributed simulations, Loki switches to an alternate form of execution, shown in Figure 7, where the Loki Graph in the host process is only used for serialization and synchronization. A group of separate processes running wtLoki work together to partition the internal solver data and evaluate the Loki node graph, including stepping the Loki Solver Setup, and communicating with MPI. This strategy lets Loki work within major host applications, using familiar data preparation and post-processing workflows, while minimizing changes needed to run distributed simulations.

5 DEVELOPER-FACING DESIGN

We now dive into the developer-facing design for the backend architecture of Loki, used by developers to build and evaluate solvers. The backend must be generic enough to describe many different kinds of solvers, such as discrete elastic rods and incompressible volumetric fluid, but also structured enough for many units of work to talk with each other in whatever way the users combine them in their Solver Setups.

5.1 Simulation Pipeline

We start by introducing the *Simulation Pipeline* and associated execution *Stages*, which are our way of representing time integration inside of Loki. Our approach shares some design elements with rendering pipelines, including a mixture of fixed functionality stages, programmable stages, and a common set of interfaces used in the pipeline.

The user-facing time step is a frame. Internally, frames are subdivided into substeps, which may be uniform or adaptive. Substeps are integrated implicitly using a Projected Newton solver, and hence typically contain multiple Newton iterations. The nested structure of frames, substeps, and Newton iterations constitutes the timeline; see Figure 8.

Tasks. The user-facing Behaviors are ultimately responsible for creating one or more *Tasks* which must be scheduled in a way that ensures proper interaction between various solvers. We try to keep the granularity of Tasks on the scale of a simple computational kernel, such as transfer particle data to grid, compute divergence of

a velocity field, or evaluate force Jacobians on the degrees of freedom. Creating rules for relative placement of Tasks associated with each pair of Behaviors onto the timeline quickly gets out of control due to combinatorial complexity. Instead, we quantize the timeline by subdividing it into Stages. The Tasks created by each Behavior are then mapped to prescribed Stages, with mapping being statically defined at development time. Maintaining consistently named Stages—and ensuring that the algorithms evaluated in each Stage map reasonably to the description of that Stage—is an imperative part of creating a successful, holistic, and flexible system.

Having too few Stages impairs scheduling flexibility, which may lead to relative ordering issues when two dependent Tasks get scheduled to the same Stage. Too many Stages becomes hard to manage, especially when it comes to introducing those to the users, in case they need to modify data within a step for art-direction purposes. We have addressed this issue by introducing a relatively small user-facing set of Stages, with the understanding they are somewhat rigid and would not change often. Generally users are able to ignore the Stages altogether, but an experienced user may employ the user-facing Stages to insert custom particle or volume expressions to run over each primitive in a Group, for added flexibility. This can be seen as analogous to programmable shaders offering optional safe places for users to customize functionality within a larger rendering pipeline. The developer-facing Stages are more malleable and more numerous, so developers can add, remove, rename, or reorder them as needed, as long as the user-facing Stages are still valid.

Figure 8 shows how the hair-wind coupling example expressed in terms of UI Behaviors in Figure 5 gets broken up into Tasks and mapped onto the set of user-facing Stages. Note that the figure shows a clear need for more developer-facing Stages: "Transfer solids data" and "Add coupled drag" Tasks are scheduled to the same Update Fluids Stage, when the data transfer needs to happen before the drag can be applied. Consequently, internally those Tasks get mapped to Data Transfer and Force Evaluation developer-facing Stages, which are ordered appropriately with respect to each other.

Newton loop. The final time integration method is eventually formed by the generated Tasks and may vary based on the Behaviors.

For elastic solids, we seek the following:

$$\boldsymbol{f}\left(\boldsymbol{x}(t), \boldsymbol{v}(t)\right) = \boldsymbol{f}_{\text{elastic}}\left(\boldsymbol{x}, \boldsymbol{v}\right) + \boldsymbol{f}_{\text{external}}\left(\boldsymbol{x}, \boldsymbol{v}\right) - \boldsymbol{f}_{\text{inertia}}\left(\frac{\partial \boldsymbol{v}}{\partial t}\right) = 0$$

That is, the total forces $\boldsymbol{f}$—including elastic, external, and inertia contributions—acting on at a location $\boldsymbol{x}$ on the body with instantaneous velocity $\boldsymbol{v}$ at time t should sum to zero. We discretize this equation of motion in time and space, and use backward Euler for temporal integration. This way, generalized positions and velocities at time t^{n+1} are connected to their previous values at time t^n as $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \boldsymbol{v}^{n+1}\Delta t$ (where $\Delta t = t^{n+1} - t^n$), such that the total force residual $\boldsymbol{f}\left(\boldsymbol{x}^{n+1}, \boldsymbol{v}^{n+1}\right)$ vanishes. Correction to $\boldsymbol{x}^{n+1}$ and correction to $\boldsymbol{v}^{n+1}\Delta t$ can be used interchangeably depending on the context. For instance, drag force derivative is naturally computed with respect to change in velocity, while elastic force derivative makes more sense with respect to the change in position. Practically, they just differ by a factor of Δt, so by convention we always use derivatives of forces with respect to positions. With these force

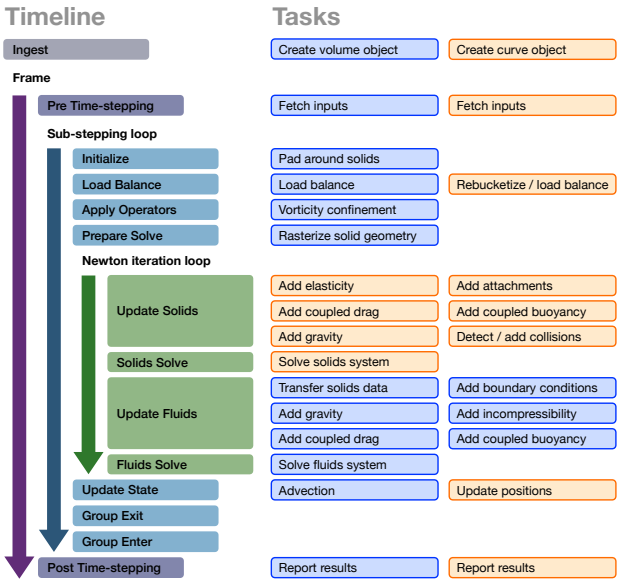


Fig. 8. A schematic representation of how wind (blue) and hair (orange) related Tasks (right) in the coupling setup from Figure 5 are mapped onto execution Stages of the timeline (left). Ingest Stage runs once at the beginning and is responsible for creating all necessary Objects and Actions, introduced in Section 5. Group Enter and Group Exit Stages are reserved for transitioning of simulation data between different groups and are not used in this example. The actual number of Tasks for our wind-hair coupling setup is much larger than presented here, as we have aggregated some of them into larger chunks of work to simplify the exposition. ©Wētā FX.

derivatives, we use Newton's method to solve for the end-of-step velocities $\boldsymbol{v}^{n+1}$. Each Newton iteration, all solid forces are assembled into a single system and solved at the same time to obtain a velocity correction. Positions are then updated to yield the next Newton iterate.

Fluids are different in that they are more commonly solved for via operator splitting, and hence position correction (advection) is separated from velocity correction (pressure, viscosity, and other solves). They are only advected once at the end of the Newton loop, so that the Eulerian representation remains fixed for the whole time step. Multiple fluids can be coupled to each other strongly via multi-phase solves, but when it comes to coupling a solid and a fluid we generally opt for a weak coupling solution. Within each Newton step we first perform a solid solve, assuming the state of the fluid is fixed. We then perform fluid-related solves, with fixed solids. To improve the stability of this weak coupling scheme, information about the solid inertia and derivatives of solid-fluid interaction forces such as drag are incorporated into the fluid system matrix as described in [Stomakhin et al. 2020]. Note that in some cases, such as rigid bodies in water, our weak coupling scheme reduces to strong coupling and can be solved with a single Newton step, similar to [Batty et al. 2007].

5.2 Intermediate Layers

Choosing an optimal scope for a Behavior is a non-trivial task. Consider a basic grid-based volumetric simulator, such as the wind

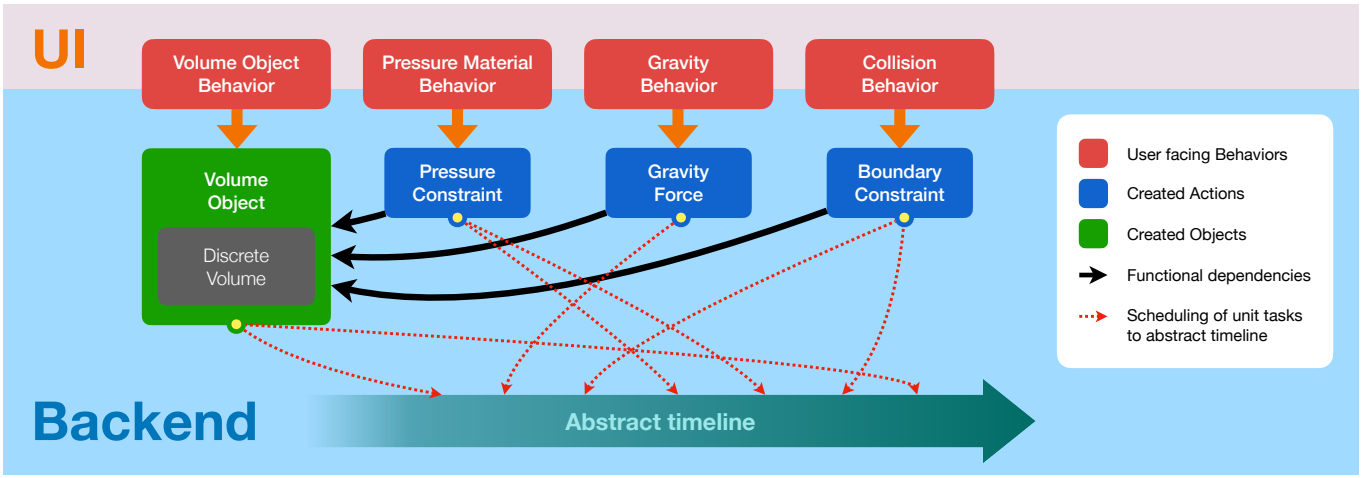


Fig. 9. A fluid solve is expressed using our user-facing set of Behaviors. Internally each behavior creates an Object or an Action with corresponding functional dependencies between the two. This translates into the required computational work that needs to be performed: for Objects and Actions to maintain and update their state and for Actions to affect Objects. The units of work are automatically scheduled to Stages of our simulation pipeline. ©Wētā FX.

component of hair in wind, as an example. Should a single behavior represent

(1) A simple computational kernel, such as divergence or gradient computation?
(2) A more complex procedure, such as a divergence-free projection?
(3) An entity that would ensure that a volume is incompressible?
(4) An entire volumetric system with collisions, forces, viscosity, and pressure solves?

Option (1) is too fine-grained for automatic scheduling, as a simple mathematical operation makes sense and may exist virtually anywhere along the timeline. This inevitably places the burden of ordering the Behaviors on the users, which goes against the goal of "a Behavior is a user-facing entity, whose role is to describe what should be simulated and not when" which is key for our Avoid Combinatorial Explosion principle.

Option (2) steps aside the explosion on number of Behaviors and alleviates the ordering issue within an algorithm unit. However, divergence-free projection could be seen in many contexts other than just volumetric simulation, and hence binding it to a prescribed Stage becomes a limitation in those scenarios.

Option (4) may seem attractive to the users, especially if equipped with an intuitive UI. However, such Behaviors tend to grow excessively large as the corresponding systems require more and more features, which becomes a maintenance nightmare for the developers. It is also unclear how to enable coupling between two such systems.

Option (3) may, at first glance, seem to be the same as Option (2). It is true that both essentially perform a divergence-free projection under the hood. However Option (3) binds it to a specific application familiar to users: enforcing incompressibility of a volume they are interested in simulating, as opposed to being a mathematical abstraction. Such formulation is not only intuitive for the users to work with, but allows the developers to unambiguously map the underlying mathematical procedure to the timeline. Thus, we deem Option (3) as the best description of a unit of work that a Behavior is responsible for performing. We refer to such units of works as Actions and introduce them more formally below.

5.2.1 Scheduling work. A Behavior has two ways of scheduling work: by creating *Objects* and/or creating *Actions*, as shown in Figure 9. These are deferred entities that are allowed to store internal state, depending on the specific Object or Action in question, and modify state using *Tasks*.

An *Object* is a physical object in space that is simulated forward in time and may store a mix of internal data representations. The Object is responsible for maintaining a consistent internal state. For instance, the Volume Object in Figure 9 would be responsible for performing advection at the end of the time step, but also for resetting the force channel before new forces are accumulated for the Newton loop. Different types of Objects are described in Section 6.1.

An *Action* is a deferred operation that operates on a single Object and, depending on the type of Action, can modify the state of the Object. We distinguish between three derived types of Actions: Forces, Constraints, and Operators, and discuss them in more detail in Sections 6.2, 6.3, and 6.4, respectively.

A Behavior may create any number of Objects and Actions it needs to perform its intended work. A Behavior is also responsible for updating the Objects and Actions it creates with new user input. These could be key framed attributes, or animated data streaming in from a file. Objects and Actions are then responsible for spawning Tasks and mapping them onto the execution Stages. Consequently, the Stages become semantically meaningful barriers inside the system, for when Objects and Actions are required to have reached a certain internal state.

5.2.2 Stage evaluation order. We now define a simple evaluation order by going through each Stage and Group and giving a callback to every Behavior, Object, and Action. See Algorithm 1. The configuration of Stages, Groups, and Behaviors are all known given any particular Solver Setup before a simulation starts; however, the number of Objects and Actions created by Behaviors are not known a priori.

ALGORITHM 1: Solver evaluation order

```
for each stage ∈ Stages do
    for each group ∈ Groups do
        if stage ∈ UserFacingStages then
            for each behavior ∈ Behaviors do
                run(stage, behavior)
            end
        else
            for each object ∈ Objects do
                run(stage, object)
            end
            for each action ∈ Actions do
                run(stage, action)
            end
        end
    end
end
```

Although all simulation-related Tasks are scheduled through the developer-facing stages, there are still reasons to evaluate Behaviors for the different user-facing stages. Dynamic settings that change during the course of the simulation must be propagated down to the Action or Object in question, and it is the Behavior that created the object that is also responsible for updating it. Additionally, users are allowed to inject custom kernels via C++ expressions that operate on the native Loki data types. This is done by creating an expression Behavior that operates on all particle or discrete volume primitives in the enclosing Group.

Objects, Actions and their derived types were not a major initial design decision for us. They emerged as a necessary intermediate layer to manage the complexity between the very high-level user-facing Behaviors and low-level Task computations. Nevertheless, they became invaluable unifying interfaces for ensuring a consistent system at a manageable level of abstraction.

5.3 Fundamental Data Types

Fundamental data types are the major storage data structures we use as building blocks to ultimately represent the internal state of any Object or Action. To achieve the Performance requirements within the Hardware constraints, we include support for distributed simulations, to solve large simulations over multiple machines using the Message Passing Interface (MPI) library for inter-process communication. Efficient support for distributed simulations is built all the way into the fundamental data types to track how data is available locally or remotely, which we detail below.

Grid Domain. The Grid Domain is a shared component between the various data objects within the Loki solvers. It maps space as a uniform grid where each cell is referred to as a *Bucket*, meaning it can contain data. Buckets are stored sparsely to avoid allocating memory for empty space. A Grid Domain can have multiple levels where each level consists of Buckets of the same size, and adjacent levels change in size by a factor of 2 in each dimension. This is similar to an octree, but can be sparse both spatially and by level. Each Bucket is assigned to be owned by a particular process leading to three subsets of Buckets per process:

- *Local subset* contains all Buckets owned by the current process. Full read/write access is available for data within these Buckets, and it is the responsibility of the current process to ensure data within these Buckets are updated on any algorithms currently running. For example, for hair in the wind, the bending constraint is only applied on vertices within the local subset for each process.
- *Neighboring subset* contains neighbors of local Buckets which are not themselves local Buckets. Read-only data access may be available for these Buckets so they are available for algorithms which require neighboring information to update the local subset; such as collision detection for hair including neighboring Buckets in the search for possible collisions when iterating over the local subset.
- *Remote subset* contains all the remaining Buckets, there is generally no data stored for these Buckets except to indicate which process owns them. Avoiding storing the remote subset data is critical to supporting simulations larger than would fit within a single machine's capacity, and significantly raises the ceiling of the scale of simulations supported.

The Grid Domain is regularly load balanced to reorganize Buckets into subsets, with the aim for each process to have the same amount of data within their local subset, and for the size of each neighboring subset to be minimized. Load balancing is generally accomplished by ordering the Buckets using a Hilbert or Morton space-filling curve, and then partitioning the Buckets based on a weighting of the amount of data within each Bucket, such as number of particles and/or voxels. Users do not directly interact with a Grid Domain, and instead work with Particle Systems and Discrete Volumes which use a consistent Grid Domain behind the scenes to track their data, see Figure 10.

Particle System. The Particle System is the main data object for storing Lagrangian data. It tracks a subset of the Buckets residing in the Grid Domain and uses these Buckets to partition particles. Each particle is represented using position and configurable channels

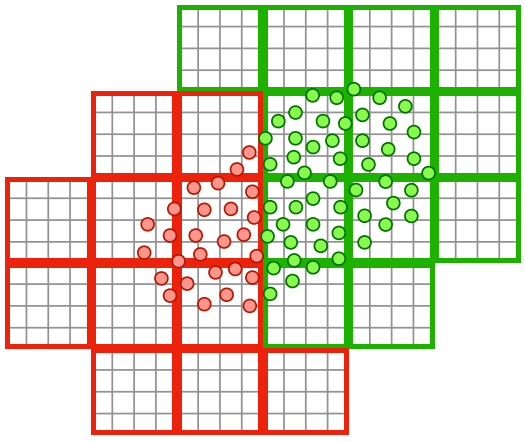


Fig. 10. A schematic illustration of a Grid Domain shared between two processes: red and green. Grid Domain tiles may be utilized by a Particle System to store particle data, or a Discrete Volume to store voxel data. ©Wētā FX.

for any additional properties such as velocity, radius, mass, index, and so on. Memory is allocated in chunks corresponding to a single channel of data for all particles within a Bucket. Operating on a Particle System is limited to updating the local subset with read-only access to the neighboring subset and dispatched per Bucket to multiple threads. This allows for work to split both locally across threads and distributed across multiple processes, while still having coherent memory access to all the particles within the current Bucket. The Particle System must be rebucketized when particles change position in order to ensure all particles reside in the correct Bucket for their current position. Currently a Particle System will only use the default highest resolution level of Buckets of a Grid Domain; however we plan to lift this limitation in the future to be able to handle spatially adaptive Particle Systems.

Discrete Volume. The Discrete Volume is the main data object for storing Eulerian data. It tracks a subset of Buckets residing in the Grid Domain, and subdivides each Bucket into a small voxel grid. The Discrete Volume stores multiple channels, where each channel is defined by a type, resolution, and sample location (cell centered or face centered). The channel resolution determines how many voxels each Bucket will be subdivided into, typically 8^3. For a single channel, the resolution within a Bucket is the same for all Buckets; but the resolution can vary between channels. Like the Particle System, the Discrete Volume allocates memory in chunks corresponding to a single channel of data for a Bucket; and work is limited to updating the local subset, parallelized by Bucket over multiple threads. The Discrete Volume uses the Grid Domain's multiple levels of Buckets to track spatially varying volumes, where the Buckets can change by a factor of 2 in each dimension between the levels of the Grid Domain. By convention, we use graded spatially varying volumes where the neighboring Buckets need to be within one Grid Domain level in either direction of the current Bucket, including corners.

A Discrete Volume together with its Grid Domain constitute an efficient volumetric data structure. While it shares many common features with other state-of-the-art sparse tiled formats such as SPGrid [Setaluri et al. 2014], OpenVDB [Museth 2013, 2021], and Bifröst [Bojsen-Hansen et al. 2021], it sets itself apart via native support for distributed computing over MPI.

Mesh Types. We have several data types specialized for tracking different kinds of mesh topologies, including Curves, Wireframe, Surface, and Body. These data types focus on tracking channels of data across geometric primitives such as vertices, faces, edges, tetrahedra, curve segments, and so on. These data structures do not make direct use of the Grid Domain and are generally used to move data in and out of the solver. Within the solver, mesh-based Objects generally contain one or more Particle Systems on which the solvers operate, but topology information is kept by additionally storing one of these mesh types. The solver will then on occasion reference the topology, such as during collision detection with meshes, or when reporting the results back onto the original mesh to return to the user.

5.4 Data Access Patterns

All algorithms are structured by sequential iteration over buckets. Any channel that needs to be read from or written to is declared using an Accessor interface. For read-write access, only the data in the current bucket can be read from and written to. For read only access, the data from the current bucket and its surrounding neighbors can be read, depending on the specified stencil pattern.

OpenMP is used to achieve parallelism over multiple threads and each process only iterates over its local subset of buckets, giving parallelism over multiple processes by design. The Accessor interface handles the logic of determining the iteration order and tracks the dependency each bucket has on its immediate neighboring buckets. This dependency information is used to schedule transfers of neighbor bucket data to and from other processes. This hides much of the complexity of distributed processes for the developer.

There are specialized Accessors for Discrete Volume and Particle System and a Multi Accessor for accessing channel data from multiple data representations when visiting a bucket. The Multi Accessor is useful for algorithms that interpolate channel data from one representation to another, such as particle to grid transfers.

6 IMPLEMENTATION

We now focus on the interface of the Objects and Actions, their derived types, and implementation of some notable examples. These components bring together varied solver components in a manner allowing them to easily interact with minimal understanding of each other, and makes them the major unifying elements of the Loki solvers.

6.1 Objects

Objects correspond to visual elements in the scene; they are the primary outputs of the Loki solver. They can be fully Lagrangian (solids), Eulerian (gas), or hybrid (particle-in-cell fluids or MPM). Loki Objects need to adhere to a common interface, specifying in particular how they are mapped to the Grid Domain for distribution and what their degrees of freedom are.

Degrees of freedom. Degrees of freedom (DOFs) are the unknowns of the system, that must be solved to compute the state of the Object at the end of the time step; they dictate the size and the contents of the final system matrix. For most Objects, DOFs are 3D quantities such as linear or angular velocities, but they can also be of arbitrary dimensions such as scalar twist for Discrete Elastic Rods [Bergou et al. 2010].

They are defined either on a Particle System for Lagrangian Objects, or a Discrete Volume for Eulerian Objects. DOFs for all strongly-coupled Objects to be solved are concatenated in a global matrix system (Section 6.5), while weakly coupled Objects are solved individually.

Embedded particles and frames. Embedded particles and frames abstract the definition of Actions from the DOFs of the objects they are expressed on, so that a single implementation automatically handles maximal coordinates, reduced coordinates, or anything in between. Objects provide semantically defined embedded particle systems, such as *boundary particles* which represent the outer shell of the Object, or *material particles* which represent its interior. Object-defined shape functions automatically handle conversions between the embedded particles and the actual DOFs, so that Actions need only consider this higher-level interface.

When necessary, Actions can also ask for emission of custom embedded particle systems. For instance, FEM-style constitutive models define quadrature points on the cells of the *material particles*, depending on the required integration order: usually a single cell-centered point for linear elasticity, or higher-order Gauss-Legendre quadrature if needed. Again, Loki will leverage the Object shape functions to transparently handle the conversions from and to the final Object DOFs.

Embedded Particle Systems may carry not only positions, but also local affine frames $\boldsymbol{F} \in \mathbb{R}^{3\times3}$ describing the material deformation gradient around those points. This is not only useful to express Constraints that depend on this deformation gradient, such as elastic constitutive models, but also to systematically associate a local orientation to particles, as is required for rigid attachment.

Notable Objects. There are many different Objects that Loki provides. We will look more closely at the Curves Object and Volume Object used in our recurring hair in the wind example.

The *Curves Object* represents the simulation data and parameters for hair simulation. Following the discrete elastic rods [Bergou et al. 2010, 2008] discretization, it handles reconstructing the curve material frame from vertex positions and edge twists. It uses two Particle Systems to store vertex-defined and edge-defined quantities, and a Curves data type to store the topology of the curves.

The *Volume Object* manages simulation data and parameters for Eulerian simulations. The data is stored using a Discrete Volume, where different channels are used to store quantities such as density, pressure, temperature, and velocity. Channels are also used to store concentrations of different materials (gases or particulates). Auxiliary channels are used to store the discretization of solid boundaries. Velocities are typically stored at face centers, and other channels are generally stored at cell centers. The Volume Object is responsible for executing emission, padding, and advection.

Other Objects. The solver framework contains a number of other Objects for different solvers, such as *Geometry Object* for elastic solids, *Hybrid Object* for FLIP liquids, *MPM Object* for MPM objects, *Particle Object* for particles and *Rigid Body Object* for rigid bodies.

6.2 Forces

In Loki parlance, a Force is a type of Action that is used to implement any force that is independent of other discretization points. This makes it the natural choice for implementing external forces such as gravity and various drag forces.

For Lagrangian Objects, all forces $\boldsymbol{f} \in \mathbb{R}^3$ and force Jacobian $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \in \mathbb{R}^{3\times3}$ are accumulated together with the Constraints (see Section 6.3) into a global linear system to be solved at each Newton iteration. By omitting the force Jacobian term for a specific force, the integration of that force effectively becomes explicit, making it possible to mix both explicit and implicit force formulations in the same simulation.

For Eulerian Objects, we use an operator splitting approach and calculate a new velocity $\boldsymbol{u}_i^\star$ for each node $\boldsymbol{i}$ and Newton iteration as

$$\boldsymbol{u}_i^\star = \boldsymbol{u}_i - \frac{1}{\Delta t}\left[\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}}\right]^{-1} \boldsymbol{f}_i.$$

This is done at the start of the Newton loop, and Constraints and Operators will get $\boldsymbol{u}^\star$ as input.

Force Density Fields. The simplest type of Force is obtained by sampling an externally provided vector volume, which can either be interpreted as force per unit mass (e.g., gravity) or per unit volume (e.g., buoyancy) depending on the settings. Various masking tools are available for the users.

Fluid Drag Force. This Force is exerted between a fluid volume and a Particle System, and depends on the relative velocity between the two. It may also depend on particle radii, orientation, and distance to the liquid surface if particles represent a liquid. The fluid volume can either be input as an externally provided volume, or from another simulated Eulerian Object in the Solver Setup. In the latter case, the forces exerted by the fluid onto the particles may be negated and splatted back onto the fluid grid to couple the two objects.

6.3 Constraints

In Loki parlance, a Constraint is a type of Action that is used to implement any implicit interaction between at least two particles, that is, any term that will induce off-diagonal coefficients in the system matrix.

Constraints can either be assembled in *stiffness mode*, contributing directly to force derivatives of the affected Objects, or in *compliance mode* through introduction of Lagrange multipliers as in [Tournier et al. 2015]. In practice we favor the former for *soft* Constraints, and the latter for *hard* Constraints, such as collisions.

Formulation. We provide facilities for assembling Constraints modeling an energy of the form

$$E = \sum_l C_l(\{\boldsymbol{x}\}_l, \boldsymbol{p}_l)^T S_l(\boldsymbol{p}_l) C_l(\{\boldsymbol{x}\}_l, \boldsymbol{p}_l) \tag{1}$$

where $C : \mathbb{R}^{3N_l} \times \mathbb{R}^{P_l} \mapsto \mathbb{R}^M$ is the Constraint *function* evaluated from a subset of particle positions $\{\boldsymbol{x}\}_l$ and a set of parameters $\boldsymbol{p}_l$ (e.g. rest quantities), and S_l are $M \times M$ scaling matrices. Under the hood, a Constraint Particle System is emitted and the user simply needs to fill, for each Constraint particle l, the channel data corresponding to the Constraint function, its gradient with respect to the Object particle positions, and optionally its Hessian and scaling matrices. Overrides are available for the rare Constraints that do not fit this formulation, though some manual assembly is then required.

As an example, the *Distance Constraint* emits one constraint particle with dimensions $M = 1, N_l = 2, P_l = 1$ for each edge of the target Curves or Mesh. The scalar-valued constraint function is defined as $C_l = \|\boldsymbol{x}_{l_1} - \boldsymbol{x}_{l_2}\| - d_l$, the difference between the current edge length and the rest length parameter d_l — usually the start-time edge length. Our embedded Particle System abstraction ensures that the Distance Constraint will work for any kind of Object defining a Curves or Mesh geometry, regardless of whether it is embedded or not.

Dynamic damping. Any Constraint using the energy formulation (1) automatically defines two damping terms with user-controllable coefficients.

- Rayleigh damping, corresponding to dissipation potential $\mathcal{V} = \sum_l \tau_l \dot{C_l}^T S_l \dot{C_l}$, with τ_l a typical relaxation time in seconds and Constraint velocity $\dot{C_l} = \frac{\partial C_l}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{x}_i}{\partial t}$ [Brown et al. 2018].
- Dahl friction, generalizing the approach of [Miguel et al. 2013] replacing the strain rate with the Constraint velocity $\dot{C_l}$.

Inverse dynamics. Many learning applications require accessing the derivatives of the physics forces with respect to user-defined parameters. Our Constraint framework automates this, requiring developers to simply provide derivatives of the Constraint function C_l, and optionally of the scaling matrix S_l, with respect to the parameters $\boldsymbol{p}_l$. Examples of such applications include:

- Compensating for gravity at the start of a simulation; for instance, in an attempt to reduce hair sagging. This can be solved in a least-squares sense as $\min_{\boldsymbol{p}} \|\boldsymbol{f}(\boldsymbol{x}^0, \boldsymbol{p})\|$, possibly with supplemental regularization. Our Constraint framework automatically assembles the matrices necessary to perform Gauss-Newton iterations on this minimization problem.
- Matching a target state. As the equilibrium state of our simulator satisfies $\boldsymbol{f} = 0$, the Jacobian of the equilibrium position $\boldsymbol{x}^{\text{eq}}$ with respect to the Constraint parameters can be computed through the implicit function theorem as

$$\frac{\partial \boldsymbol{x}^{\text{eq}}}{\partial \boldsymbol{p}} = -\left[\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right]^{-1} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}}.$$

  Evaluating the Jacobian in the target residual direction (i.e. performing a back-propagation step) thus roughly amounts to one forward Newton iteration.

Attachment Constraint. Following the Avoid Combinatorial Explosion principle, we want our Attachment Constraint to work on any pair of Objects, regardless of their topology, and avoid having to write specific code for any kind of coupling. To achieve these goals, we separate the definition of a Constraint Behavior into two Stages, *Emission* and *Response*.

The Emission Stage specifies which points on a set of Objects should be attached together. The user may provide explicit attachment networks (low-level workflow), or the Constraint may generate them automatically using closest-point or all-within-range queries (high-level workflow). For the latter, we only need to implement these spatial queries once for each kind of boundary topology (surface, curves, points). The user may also choose to emit Constraints only once at the start of the simulation, or continuously at each simulation time step.

The Response Stage must allow modeling of any kind of Constraint, from maintaining distance only to fully rigid, regardless of which kind of Objects are attached together. For this, we leverage the embedded boundary Particle System frames $\boldsymbol{F}$ and denote the interpolated frames at the source and target points as $\boldsymbol{F}_{\text{source}}$ and $\boldsymbol{F}_{\text{target}}$, respectively. For rigid attachment, we want to maintain the relative displacement $\boldsymbol{d} = \boldsymbol{x}_{\text{source}} - \boldsymbol{x}_{\text{target}}$ in the target frame $\boldsymbol{F}_{\text{target}}$; however, for pure distance-preserving constraints, this would introduce an unwanted linearization. To handle both cases, we define a new *attachment frame* $\boldsymbol{A}\left(\boldsymbol{d}, \boldsymbol{F}_{\text{target}}\right)$ that interpolates between the target frame and a displacement-aligned frame based on a *flexibility* coefficient ξ. For $\xi = 0$, $\boldsymbol{A}\left(\boldsymbol{d}, \boldsymbol{F}_{\text{target}}\right)$ coincides with $\boldsymbol{F}_{\text{target}}$, while for $\xi = 1$, the first axis of the attachment frame is fully aligned with $\boldsymbol{d}$. Equipped with this new attachment frame, we define our Constraint function C as the concatenation of four 3D terms $C^0 \dots C^3$,

$$C^0 := \boldsymbol{A}\left(\boldsymbol{d}, \boldsymbol{F}_{\text{target}}\right)^T \boldsymbol{d} - \left(\bar{L}, 0, 0\right),$$
$$C^k := \boldsymbol{A}\left(\boldsymbol{d}, \boldsymbol{F}_{\text{target}}\right)^T \boldsymbol{F}_{\text{source},(k)} - \bar{\boldsymbol{a}}_k \qquad \text{for } k = 1 \dots 3,$$

with $\bar{L}$ the constraint rest length and $\bar{\boldsymbol{a}}_k$ rest orientation parameters.

We can then play on the flexibility parameters ξ and the scaling matrix S to adjust the behavior of the Attachment Constraint. To reduce the number of exposed parameters, we typically use

$$S := \text{diag}(k_N, k_T, k_T, \beta_1, \beta_1, \beta_1, \beta_2, \beta_2, \beta_2, \beta_3, \beta_3, \beta_3).$$

- For $\xi = 0$, the Constraint will maintain the relative position of the source frame in the target point, and the strength of the normal and tangent responses can be adjusted with k_N and k_T, respectively.
- For $\xi = 1$, the Constraint will maintain the distance between the two attached points, and k_T will be without effect.
- For $\beta_k > 0$, the Constraint will also maintain the relative orientations of the two Objects. This is especially interesting for attaching rigid bodies, or the root of a hair strand to the scalp to limit twisting.

Collision Constraint. Similar to the Attachment Constraint, we want our Collision Constraint to work on any pair of Lagrangian Objects, regardless of their topology. Again, we separate the Constraint workflow into two passes, *Detection* and *Response*. The Detection pass may use one or multiple algorithms among point-signed distance field (SDF), mesh-based continuous-time, and lattice detection [McAdams et al. 2011]. Combining those algorithms can be beneficial, for example mesh-based continuous-time collision detection will be more precise, but SDF-based detection will offer more robust depth information. Collision detection is accelerated by leveraging the Bucket-structure nature of Loki Objects: only primitives living in the neighboring Buckets are considered as potential collision candidates. This greatly reduces the amount of communication necessary between machines, but imposes a limit on the maximum relative velocity between colliding Objects. In any case, the Detection pass generates a set of Constraint particles with coordinates of contact points, contact normal, and target separation distance, which will be consumed by the Response algorithm.

For the Response pass, two algorithms are again available and can be combined. The first is implicit penalties; in this case, we simply define the Constraint function as the concatenation of the penetration distance and the tangential displacement clamped with a Coulomb-like law. The other more commonly used response option is hard collisions; we follow the method from [Daviet 2020] and extend it to support rotational degrees of freedom (as with rigid bodies) by falling back to the local Coulomb friction solver from [Daviet et al. 2011]. Efficient distributed solves are achieved by sorting contacts depending on whether they are fully local, involve a foreign particle, or involve a particle touched by a foreign contact, and focusing on local computation while cross-machine transfers happen.

6.4 Operators

For Actions that do not fall into Force or Constraint categories, Operators provide a way to modify simulation state without affecting the coupled system for solids or weak solve for fluids. They can be used to achieve state transitions on Objects (e.g., the emission, absorption, merging, and splitting of fluid particles), apply external and artistic controls (e.g., vorticity confinement, fluid targeting, and partial kinematic guiding), or run secondary solves (e.g., SPH and reduced flow solves). Despite greater freedom of Operators compared to Forces or Constraints, it is the responsibility of each Operator itself to handle interactions and coupling across Objects and ensure conservation of mass, linear and angular momentum, and so on.

Vorticity Confinement Operator. The Vorticity Confinement Operator adds a vorticity confinement correction term to each of the affected discrete volumetric velocity fields on every simulation step, as described in [Fedkiw et al. 2001]. This models the small scale rolling features of fluids that are absent on most coarse grid simulations.

Coupled Buoyancy Operator. The Coupled Buoyancy Operator defines the interaction between a Lagrangian Object and an Eulerian fluid, which can either be represented as a simulated Hybrid (FLIP) or Volume (Eulerian) Object, due to the shared pore pressure as described by [Stomakhin et al. 2020]. The Lagrangian Object acts as a collider in the pressure projection. Optionally, a compliance term may be added to account for its inertia. This makes it possible to deal with a variety of coupling situations from hair in the wind to rigid bodies floating on the ocean surface.

6.5 Linear Equation Solvers

Both fluid and solid implicit updates within our Newton step boil down to systems of linear equations. To put it in Loki terms, we would want to choose a linear solver depending on the kind of Force or Constraint being simulated. Mixing multiple Objects, Forces, and Constraints, however, and coupling them strongly complicates things considerably, as then the system matrix may potentially contain sub-blocks with drastically different properties, and the choice of linear solver becomes non-trivial.

We encounter a tension between the overall Performance goal and the Avoid Combinatorial Explosion design principle, which when taken to the extreme leads to writing specialized optimal solvers for each combination of behaviors or a single linear equation solver for all use cases respectively. The former option was not a sustainable scope of work. The latter was also found to not be sufficient as solving such a wide variety of systems has been studied extensively in the linear algebra literature [Golub and van Loan 2013], and there is no one-size-fits-all solution for the range of systems and scales encountered in production. We settled for a middle ground with a handful of custom linear equation solvers which provide excellent performance on our most common use cases while still maintaining wide coverage of solid performance both locally and distributed. Below we present the collection of linear solvers that Loki implements and explain when each of them is used. On a rare occasion when our linear solver of choice fails, we would use PARDISO as a potentially slower, but a fail-safe option.

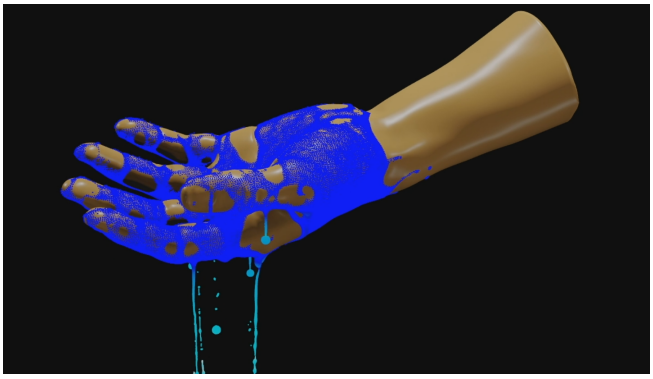


Fig. 11. **Thin film.** Example frame of a thin film simulation on a skin surface following [Stomakhin et al. 2019]. ©Wētā FX.

ALGORITHM 2: Predictor-Corrector LCP Solver

Input
A system of linear equations from mixed LCP
$Ax + b = w = w_+ - w_-$
$0 \le u - x \perp w_- \ge 0$
$0 \le x - l \perp w_+ \ge 0$
Predict
$A'x' = b' \leftarrow Ax = b$ without constraints
Solve $A'x' = b'$ with Preconditioned Iterative Solver
Correct
Set initial guess of PGS as x'
Solve $Ax = b$ with PGS
return x

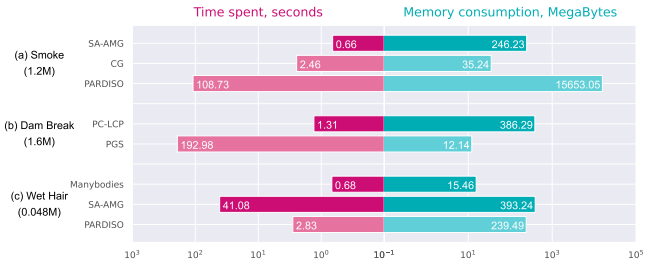
Smoothed Aggregation Algebraic Multigrid (SA-AMG). Pressure and viscosity solves for fluids typically lead to large sparse systems of linear equations; and Krylov subspace methods, such as Conjugate Gradient (CG), are known to perform well in those scenarios. In its pure form CG demonstrates poor convergence properties, and generally requires a preconditioner to achieve acceptable performance results [Barrett et al. 1994; Young 1971]. Multigrid methods offer by far the best preconditioning for Poisson-like problems [McAdams et al. 2010]. To avoid the overhead of tuning Geometric Multigrid for every specific problem at hand, we opt for implementing an Algebraic Multigrid (AMG), which adjusts automatically depending on the properties of the system matrix. Specifically, we implement Smoothed Aggregation (SA) AMG as a preconditioner for CG [Demidov 2020; Janka 2007; Tamstorf et al. 2015] and use it for all of our fluid updates. Figure 12(a) shows a benchmark test of SA-AMG for a simple smoke scene. We compare our SA-AMG solver to CG without preconditioner and sparse direct solver(PARDISO) for reference. SA-AMG gives much better convergence rate and reduction of computing time over CG. Also the test shows that the iterative solvers perform better than the sparse direct solver for this kind of problem.

Predictor-Corrector Linear Complementarity Problem solver. Enforcing non-stick Neumann boundary conditions inside the pressure solve for fluids results in a mixed Linear Complementarity Problem (LCP). There are many methods for solving LCP; the most famous and widely used one is Projected Gauss-Seidel (PGS) [Bender et al. 2014; Silcowitz et al. 2010]. PGS works successfully for many applications, but for large problems the convergence rate is still rather slow [Young 1971]. Instead, we have developed a simple

predictor-corrector technique based on the idea that the solution of a system that ignores the constraint will be close to the solution of the original LCP. As outlined in Algorithm 2, we apply a robust preconditioned iterative solver, typically SA-AMG, as a predictor and set the solution as initial guess for PGS. As the initial guess is close to the solution, PGS now converges quickly. Figure 12(b) shows the result of a benchmark test on a simple dam break simulation with non-stick fluid boundary condition. The result shows that our simple technique saves compute time considerably, with much better convergence rate compared to PGS alone.

Manybodies iterative solver. Stiff Forces and Constraints for many solid objects is another important use case which encompasses hair, plants, muscles, and so on. For a small number of DOFs, direct solvers are generally preferable, as iterative ones may experience convergence issues. However, as the number of DOFs increases direct solvers quickly become inefficient as well. We thus introduce a custom solver for such large stiff problems, which we call the *manybodies iterative solver*, since it is typically used for setups with multiple interacting solid objects.

Figure 13(a) shows the characteristic matrix structure that a system consisting of multiple objects creates. Diagonal sub-blocks correspond to Objects, and additional off-diagonal terms may be present because of Collisions or Attachment Constraints. We start with applying a direct solver to each of the diagonal sub-blocks, ignoring off-diagonal connectivity, as shown Figures 13(b) and 13(c). We then use this procedure as a preconditioner for the full system inside an iterative solver, typically CG. This approach gracefully handles different material properties of individual objects independently (which allows for trivial parallelization) within the preconditioner, and leaves Collisions and Attachment Constraints resolution to the



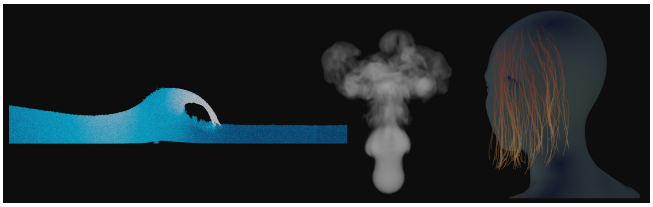


Fig. 12. Runtime and memory results for benchmark tests of linear equation solvers. The numbers besides each benchmark indicate the size of the system of linear equations. (a) Smoke benchmark compares our SA-AMG, CG without preconditioner, and popular sparse direct solver PARDISO on a smoke plume. (b) The Dam Break benchmark compares PC-LCP to PGS on a LCP dam break with non-stick fluid boundary conditions. (c) The Wet Hair benchmark compares our Manybodies solver, our SA-AMG, and PARDISO on a wet hair simulation. ©Wētā FX.

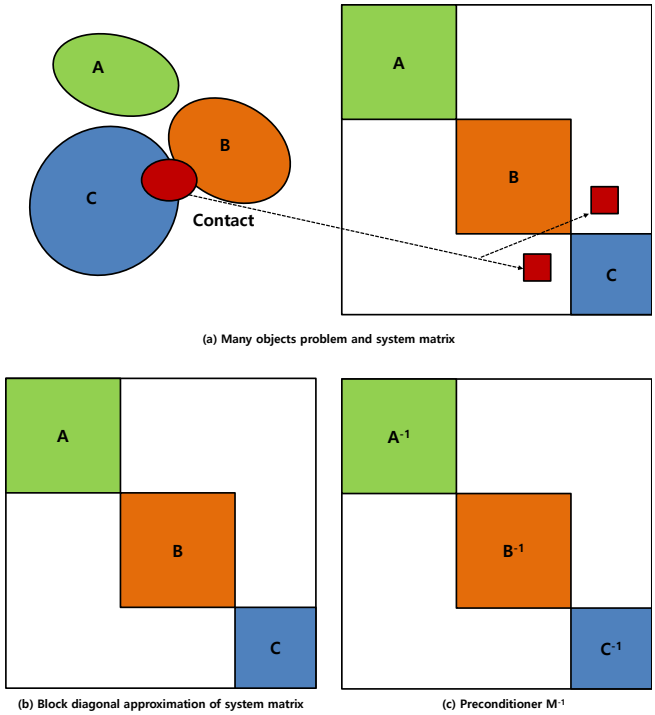


Fig. 13. The manybodies problem creates a block structure where: (a) Each Object creates a sub-block. If there are Constraints between Objects, then connectivities are added to the blocks. (b) The manybodies iterative solver creates sub-blocks and simply ignores the connectivity between blocks. As connectivity won't be a major component of the system matrix, the diagonal blocks will be a good approximation of original matrix. (c) The manybodies iterative solver provides the inverse of each sub-block independently. As sub-blocks are quite small compared to the entire system matrix, applying a direct solver for each sub-block is efficient. ©Wētā FX.

encompassing iterative procedure. Figure 12(c) shows a performance comparison between our manybodies iterative solver, our SA-AMG solver, and the MKL PARDISO sparse direct solver.

The manybodies iterative solver can be regarded as a variant of domain decomposition [Boxerman and Ascher 2004] and prefiltering methods [Eberle 2018; Kim and Eberle 2020; Tamstorf et al. 2015]. The main difference is that a block is created per Object, rather than using a prescribed size or graph partitioning. The blocks are then used as preconditioning matrices. Thus, a typically small number of intricately tangled DOFs within as single Object, such as a FEM mesh, are resolved with a direct solver. Constraints on the other hand lead to a sparser inter-block connectivity, which is readily resolved with an iterative solver.

7 APPLICATIONS

We now describe five applications of Loki to specific physics-based animation problems. These applications are representative of work required by technical artists for elements in major film or TV production. All would generally require multiple distinct passes and/or multiple artists, but can be done by a single Loki user in a single pass. Table 1 shows performance numbers for the applications below and other examples in the paper.

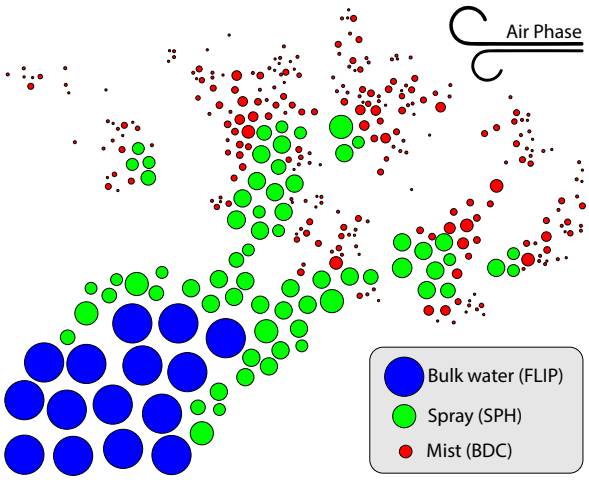


Fig. 14. A breakdown of the water states we use in our state machine approach for multi-scale water. Particles split and transition as they move from bulk water to spray to mist. The methods used in each state are optimized for the respective scale. ©Wētā FX.

7.1 Boat Wake

Our first application demonstrates the power and versatility of using our declarative configuration to define a state machine for boat wake simulations. Simulating a boat wake presents a challenging scenario because it involves water behavior at multiple scales simultaneously. This behavior can be broken down into three components or states that are represented as Groups, as described in Section 4.1:

(1) Bulk water: The large-scale motion of water as it collides with a fast-moving boat.
(2) Spray: Thin features like tendrils and large irregularly shaped water droplets that split off from the bulk water.
(3) Mist: Fine droplets forming from the spray which are highly affected by air drag.

While a very high-resolution FLIP water simulation could resolve each of these states, it would be prohibitively expensive to do so. This problem is exasperated in production scenarios where a large number of boat wake simulations are needed on a tight schedule. We apply the Best-in-Class design principle to note that different simulation methods even within the same simulation would be advantageous, so we simulate each of these states within the same simulation but with different simulation methods that maintain high fidelity while being much more efficient for their respective scales and behaviors.

We simulate the bulk fluid with FLIP, spray with SPH, and mist as ballistic particles that can collide and coalesce or separate. We use a method we call *binary droplet collisions* (BDC) based on [Jones and Southern 2017] to resolve ballistic particle collisions. This results in a much more natural distribution of particle radii as the simulation progresses. A breakdown of these states and methods is shown in Figure 14.

Loki's declarative configuration paradigm makes this setup straightforward. The bulk water is represented with a Hybrid Object with a Pressure Material Behavior to ensure incompressibility. The spray is represented as a Particle Object with an SPH Behavior. The mist is represented as another Particle Object with a BDC Behavior. Finally, we also create a Volume Object to represent air and use a two-way coupled Drag Force Behavior to capture interaction of spray and mist with the surrounding air.

7.1.1 State transitions. With the behavior of each state defined, we now describe how we transition particles between the states in two parts: our transition criteria and our particle up-resolution process.

Our transition metric is based primarily on speed, distance to the fluid surface (bulk water to spray only), near-neighbor count, and pressure gradient magnitude. Using a Particle Expression Behavior, these quantities can be computed and compared to a user-defined threshold to determine if they will transition from bulk water to spray or from spray to mist. We have found that the pressure gradient magnitude is usually the best indicator for transitions and generalizes well across many different scales and scenarios. Intuitively, if the magnitude is small the fluid is in free fall, and hence transition from bulk fluid to spray is appropriate.

As we transition from bulk water to spray to mist, we split the particles into smaller ones to provide more resolution and fidelity as we progressively move to less expensive simulation methods. Because FLIP simulations typically have a uniform particle radius, we initialize split particle radii using an inverse cubic distribution tuned by the user. The radius and spread of these splits can also optionally be weighted based on the transition metric; for example very fast moving particles with a very low pressure gradient magnitude would split into smaller, more frequent particles that are further apart than particles that barely exceed the transition threshold. This method combined with BDC avoids unnatural uniform distributions as the particles become ballistic. When mist or spray particles re-enter the fluid surface of the bulk water, their radii are adjusted and they are transitioned back into FLIP particles.

Our state machine approach is demonstrated in the boat wake example shown in Figure 15. This result was generated automatically using Loki with very little upfront tuning and no post-processing. This demonstrates that Loki can be used to quickly generate high-fidelity water simulations without requiring senior technical artists.

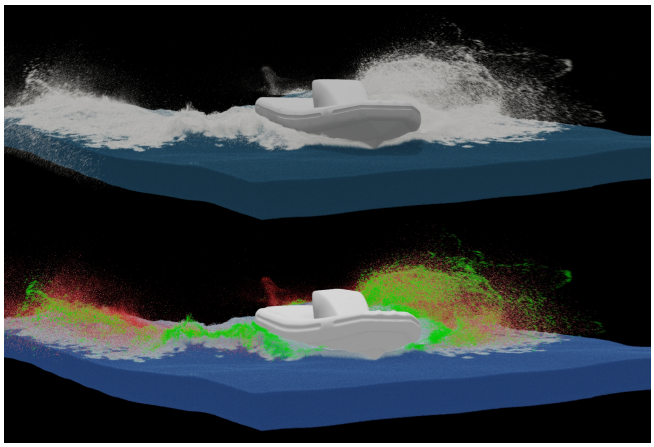


Fig. 15. **Motor boat.** Example frame of a boat wake simulation that uses our state machine approach (top) is shown with a breakdown of the water states (bottom). As indicated in Figure 14, bulk water is shown in blue, spray in green, and mist in red. ©Wētā FX.

Fig. 16. **Palm tree.** A tree represented with elastons coupled with a wind volume blowing from left to right. ©Wētā FX.

Furthermore, we have found that this approach has simplified our production review process. Without this method, a time-consuming approval was required at each stage of water splash simulation: one for the bulk fluid, another for the secondary spray, and more for any additional mist passes. With our state machine approach, only one review is typically needed because each of the states is successfully simulated simultaneously. This is all done at a fraction of the computational cost of simulating a comparably high-resolution FLIP simulation and is demonstrative of the flexibility beyond traditional coupling, using the many solvers available within Loki.

7.2 Windy Tree

Our next example shows the modularity of the Loki architecture by simulating a tree in the wind based on a minimal number of changes from our previous hair in the wind example, see Figure 16.

Plants often have a very high geometric complexity at multiple scales ranging from tiny stems to massive trunks. Efficiently capturing that geometry can lead to a range of dimensionality required, such as thick trunks with 3D geometry, large leaves with 2D surfaces, and long branches with 1D curves. We have found elastons [Martin et al. 2010] to be an excellent model for plants due to the ease it represents a variety of dimensionality within the same object.

In keeping with our Avoid Combinatorial Explosion, we only need to make a few changes starting from our hair in the wind Solver Setup to simulate a tree as elastons coupled with the wind, as shown in Figure 17:

- The Hair Group is changed to a Plant Group by updating the Curves Object to a Geometry Object to work on general geometry instead of only curves.
- The Stretching Constraint and Bending Constraint are removed as they were tied to the Curves Object to represent discrete elastic rods.
- An Elaston Constraint is added to assign the codimensional elaston model to the Geometry Object.

Within the Plant Group, the Attachment Constraint and Collision Constraint already work for all kinds of geometry, and therefore do not need to change. The Wind Group contains everything needed to simulate the wind independent of any geometry it acts on, and hence also remains unchanged. Finally, the Root Group Behaviors, including Gravity Force, Coupled Buoyancy, and Coupled Drag Force, work on all kinds of geometry and remain unchanged as well.

The ease of changing individual components, such as the geometry representation, while maintaining the same functionality and interactions with other components greatly eases the configuration burden on the user by helping them adopt Best-in-Class models without significant changes to similar familiar setups.

7.3 Super Solver Setup

Our next example demonstrates how we leverage the flexibility and modularity of our framework to reduce setup burden for our users. We present two scenes involving elastic objects (see Figure 2): one involves a fluffy elastic bunny settling down for a nap on a hammock; and in the other, an animated character, complete with hair and clothing, leaps from a diving board into a pool. The first example shows strong coupling between 1D, 2D, and 3D elastic solids through frictional contact and rigid attachments (described in Section 6.3). Note that the bunny's degrees of freedom are located on an enclosing tetrahedral lattice, whereas attachments and collisions are resolved on the surface mesh. The second simultaneously deals with free-surface fluids, elastic solids, and the coupling in between.

For both of these examples, we use discrete elastic rods [Bergou et al. 2010, 2008] to simulate the hair and fur. Our thin shell model is based on Volino et al. [2009] and Grinspun et al. [2003], which we use to simulate both the cloth and the diving board. We simulate the tetrahedral lattice enclosing the bunny using a linear FEM model with the Stable Neo-Hookean constitutive model [Smith et al. 2018]. The frictional contact between each of these structures is reliably handled by the recent technique from Daviet [2020]. In the second example, we simulate the water using a FLIP method [Zhu and Bridson 2005] enhanced with buoyancy and drag forces on the elastic solids [Stomakhin et al. 2020]. Additionally, the air is treated as an incompressible fluid and therefore contributes to the solids' dynamics through two-way coupled drag force. Moreover, we have wet hair and wet cloth solver components, based on [Fei et al. 2017] and [Fei et al. 2018] respectively, which model the internal fluid flows and the resulting cohesion on the hair and cloth, and transitions to/from external fluids through water absorption and dripping. We

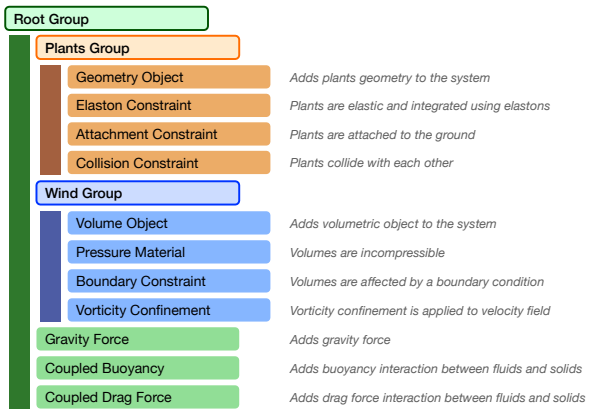


Fig. 17. Solver Setup for elaston plants coupled with a volumetric wind. Minimal changes are needed from the hair in wind Solver Setup demonstrated in Figure 5. ©Wētā FX.

Fig. 18. **Brasier fire.** Example of a fire simulation. ©Wētā FX.

opt for a weak coupling model between the solids and fluids, as described in Section 5.1. This way, both scenes are completed within a single simulation pass.

Despite the two scenes being quite different in nature and comprising a number of interacting solvers, we use the same conceptual simple user-facing Solver Setup for both simulations. In our Behavior tree, we have four Groups dedicated to individual phenomena: elastic rod simulation, elastic shell simulation, elastic solids, and fluids. Behaviors related to coupling, such as collision, attachments, drag, buoyancy, etc. appear outside the four groups. In practice, we construct a Solver Setup like this once and reuse it for all scenes featuring characters or elastics. If no fluid appears in a scene, all the fluid-related Behaviors (including fluid-solid coupling) are automatically disabled; similarly for rods, shells, and elastic solids. Material parameters can be set from the host application to override default values encoded within the Solver Setup. This workflow makes it easy and efficient to set up new scenes, regardless of the complexity and number of interacting phenomena. As the semantic meaning assigned to each of our Behaviors generally remains fixed, Solver Setups rarely require updates, even as we regularly add new features.

7.4 Explosions and Fire

The Eulerian fluid solver in Loki is based on the common method of operator splitting [Stam 1999]. In addition to the typical fluid dynamics components (pressure solver, viscosity solver, and advection) it has a number of components for thermodynamics, chemistry, turbulence enhancement, and artistic control. These components can be combined in the Solver Setup to produce a number of different volumetric phenomena, such as combustion (fire and explosions), smoke, dust, mist, formation of clouds, and air fields for coupling with other solvers. The initial shape of an explosion is modeled using a lightweight procedural particle animation in Houdini. The particles are converted to a volumetric representation that includes fields for fuel, temperature, and velocity. These volumetric fields are used as emission volumes, and these are only present for the initial 10-20 frames. After that the volumetric solver takes over and evolves the explosion.

Emission is either done using *stamping*, which sets values directly based on user-provided volumes, or using a *fluxed* approach that sets a boundary condition for pressure solver and advection. Prior to emission, the Volume Object creates Buckets based on the bounds of the emission volume. Additional Buckets can be created based on the Bucket subset of other Objects.

Padding adds layers of Buckets around the simulated quantities, to ensure that there exist voxels to advect into; or to allow the simulation, if required, to capture fluid flow in a wider region than the domain of immediate visual interest. There are typically 3-4 layers of Buckets surrounding a flame front or smoke plume, as more than that is not computationally feasible for simulations with a large Bucket count (>1M). A common way to increase world-space size of the padding layer, if needed, is through the use of spatial adaptivity.

Advection is done using the typical tracing approach, and supports a number of different schemes, such as Semi-Lagrangian, MacCormack and BFECC [Selle et al. 2008], and also various higher-order interpolation schemes. Dissipation reducing time integration methods such as Advection-Reflection [Narain et al. 2019] are also available.

The volumetric fluid solver is designed to handle large simulations by distributing the simulation across multiple machines using MPI. A typical production quality simulation of a large scale explosion requires simulating with a voxel size of 2-5cm, which can result in millions of Buckets of volumetric data and billions of voxels. This may not fit on a single machine, and often needs to be divided over 4-8 machines.

Most of the volumetric algorithms only depend on data from the current Bucket or the immediate neighborhood of Buckets. Tracing stands out, in that it is completely dependent on the velocity magnitudes. Large velocities could lead to traces that end up further than the immediate Bucket neighborhood. A conservative approach would be to enforce a strict CFL condition, but this is not practical for simulations with fast-moving emitters or colliders. In order for advection to scale well on MPI, we classify traces as local or foreign depending on whether they end up in a Bucket that is owned by the local process or a different process. We then bundle traces based on the destination process, and transmit the trace data to other processes. Once transmitted, each process interpolates advected quantities using local data. Finally, the interpolated quantities are transmitted back to the process where the traces originated.

The volumetric fluid solver supports spatial adaptivity by varying the size of the Buckets. Spatial adaptivity has a few typical use cases:

- Use a wide padding of successively coarser Buckets around an explosion or smoke plume to be able to capture more of the flow of the surrounding air. This produces more realistic shapes and would not be feasible when simulating with uniformly sized Buckets.
- Adjust the size of the Buckets based on simulation quantities, e.g. use the finest resolution near flame front, intermediate resolution for smoke, and the coarsest one away from regions of visual interest.
- Adjust the size of the Buckets based on a camera frustum, where the size of the Buckets increases with the distance from the camera position.

The implementation needs special consideration at the border between Buckets of different sizes and is mainly based on the methods described in [Ando and Batty 2020]. Examples of an explosion and a fire produced with our solver are shown in Figures 1 and 18.

The Solver Setup for explosions can be extended with additional Groups of one-way or two-way coupled particle systems for soot particles and embers. This avoids having to set up and run additional passes for these kinds of phenomena.

7.5 Foam and guided bubbles

Finally, we show a production example of foam and bubbles, commonly referred to as *whitewater*. The method is based off [Wretborn et al. 2022] which separates the simulation technique into two separate solvers. Particle based bubbles are two-way coupled with a sparse, Eulerian fluid with interaction defined as a force exchange of buoyancy and drag. The fluid is guided from a pre-cached fluid simulation, ensuring it can be run as a secondary process. Foam is simulated as a viscous fluid using SPH for discretization constrained to the fluid surface, approximating thin and wet foam. Interactions with foam and bubbles are implemented as *transitions*: momentum is exchanged by moving particles between the two simulation methods based on their proximity to the fluid surface. An example can be seen from Figure 19.

In total there are 3 separate Objects in this system (bubbles, foam, the sparse fluid) that all interact weakly. The resulting Solver Setup complexity is similar to that of Figure 5, with the difference being one more Group to represent the foam.

7.6 Additional fluid examples

Figure 11 shows a thin film water simulation following [Stomakhin et al. 2019]. Figure 3 demonstrates coupling of a FLIP liquid with the surrounding Eulerian wind to produce a large scale waterfall.

8 DISCUSSION

We have described our proposed framework, and we have shown that it can successfully be used to simulate a diverse set of applications. We will now detail the strengths and weaknesses of our method based on our experience using the system in production for approximately four years.

8.1 Strengths of Loki

There are a number of ways in which Loki's design decisions have played to the framework's advantage.

Single Consistent System. Loki's design means that we can simulate a wide range of phenomena within a single framework, including liquids, smoke, fire, explosions, hair, muscle, cloth, feathers, plants, and rigid bodies. There are many benefits to this approach. By having a single solution for simulations of all kinds, we can reuse common data structures and techniques. Implementing a feature or improvement to one component, such as collision handling, benefits several different domains simultaneously, with only modest overhead. Conforming to Loki's guiding principles (adherence to physical units in particular) means that little effort is needed to introduce coupling between any pair of domains.

No Compromises on Quality. Compared to dedicated solvers, Loki delivers qualitatively equivalent results. Despite its requirements

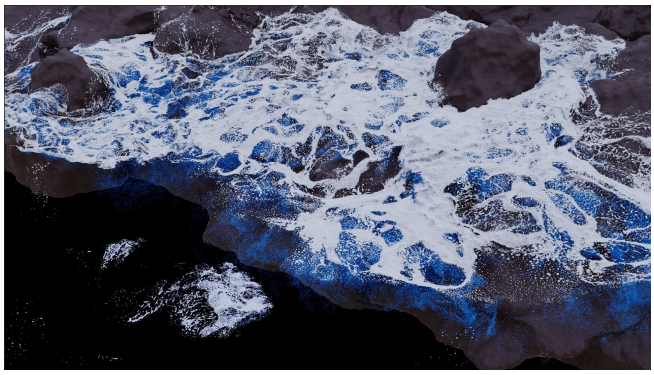


Fig. 19. **Rocky shore.** A secondary simulation of foam (white) and bubbles (blue) on a pre-cached fluid simulation (not shown). ©Wētā FX.

on structure, it is rare that a state of the art method cannot be effectively implemented within these bounds. Physical fidelity can be obtained in all domains without one domain enforcing burdensome restrictions on another.

Transitions between Representations. Loki grants the ability to abstract beyond a single representation of a material by dynamically transitioning from one domain to another while preserving properties such as mass, energy, and momentum. Through these transitions, we can choose the discretization that best suits the needs of a particular scene; for example, bulk fluid simulated in a large Eulerian domain may be transformed into Lagrangian particles to capture highly detailed splashes and droplets around colliders.

Modularity. Because of its structure, Loki allows developers to experiment with, deploy, and publish novel advancements within the larger system. These improvements can immediately impact multiple domains. For instance, novel geometric representations can be added without significant changes to constitutive models, collision algorithms, attachments constraints, coupling algorithms, and so on. Additionally, the Solver Setup modular declarative design makes it easy to reuse Groups, such as starting from a dry hair Solver Setup as the Hair Group for a wet character simulation.

8.2 Areas for Future Improvement

Loki's design decisions and requirements in some areas lead to the following compromises. We hope to explore ways to reduce the scale of these drawbacks as part of future development.

Development Burden. Because of its strict structures and requirements, Loki is not well suited for rapid prototyping by simulation developers. Additional development overhead is inherent to adding new components compared to a standalone solver. For this reason, when experimenting with novel concepts, we often test prototypes externally before porting them to Loki.

Memory Overhead for Solids. There is some additional memory overhead to generically handle 1D (hair), 2D (cloth), and 3D (muscle) geometric elements together. It is common to represent 3D solids using positional degrees of freedom, but 2D solids may need a separate degree of freedom dimension for the thickness. Similarly, the discrete elastic rods representation, which we use for 1D hair, has to represent twist along edges in addition to node positions.

Table 1. Simulation times, resolutions, and settings for some of the examples presented in this paper. Numbers of particles, voxels, and time per frame are reported for representative frames that correspond to the ones shown in the figures.

Example	Particle count	Voxel count	Steps per frame	Newton iterations	Time per frame	Memory	Machine
Palm tree	335K	1.38M	2	1	4s	6.1GB	24 Cores
Thin film drips	1.37M	5.79M	100	1	5m 18s	4.4GB	24 Cores
Interacting solids	950K	-	2	2	3m 49s	6GB	24 Cores
Brasier fire	-	488M	2	1	24m 28s	63GB	24 Cores
Swimming character	8.8M	35.3M	3	2	6m 10s	57GB	52 Cores
Waterfall	14.2M	173M	2	1	7m 56s	105GB	52 Cores
Motor boat	117M	38.9M	6	1	1m 42s	99GB	64 Cores
Rocky shore	4.0M	18.4M	4	1	1m 28s	35GB	64 Cores
Explosion (MPI)	-	957M	5	1	10m 28s	4 x 60GB	4 x 24 Cores

We address this disparity by having any node of any dimension represented by both a position and an affine coordinate frame. This gives us uniformity in representation across different kind of solids. However, these additional coordinates correspond to more degrees of freedom than would be needed for a specialized simulator, which can result in additional overhead for storage and degree of freedom reduction (e.g., reducing an affine frame to rod twist) during system assembly.

Performance Overhead for Solids. We currently require neighboring degrees of freedom to live within a single Bucket neighborhood of the grid domain. This imposes a minimum Bucket size in our scene relative to the scale of the longest edge in our geometry. Because we parallelize most operations over Buckets, this has performance implications: large Bucket sizes imply poor parallelization. To address this, we would like to explore ways to decouple the domain and geometry scales, as well as better domain decomposition control for elastic solids.

MPI Requirements. In order to support distribution across multiple machines, all algorithms need to respect a set of MPI-related rules such as limiting data access to local neighborhoods and careful synchronization, which puts additional burden on the developer.

Parameter Complexity. Loki's feature set has grown very large, which has resulted in a large number of parameters; this can sometimes be overwhelming when users are trying to understand how to guide specific adjustments from the default results, with so many interacting components. We would like to explore ways of offering optional additional layers of simplified controls for users who want a reduced interface. We would also like to investigate selecting more parameters automatically; for example choosing the best linear equation solver, or the ideal Bucket size.

8.3 Areas for Future Exploration

In addition to ameliorating these drawbacks, we intend to extend Loki in several directions to make it a more capable and modern simulation solver. Future work includes incorporating more solvers into Loki's framework, such as an atmospheric cloud simulator to ensure that Loki remains up to date with the state of the art, as simulation fidelity improves. In order to improve the performance of our simulations, we would like to employ wider use of spatial adaptivity for fluids and varying topology for meshes, reducing the amount of data to be processed in regions where high resolution is not necessary. Detecting and resolving collisions between all kinds of geometries and representations is a bottleneck in many simulations, especially those that are heavy in collisions; we would like to explore ways to expedite this step of the solver. Due to hardware limitations in our production environment, we have steered away from incorporating GPU programming into Loki; but in the future we would like to explore how highly parallel devices could be used within the Loki paradigm. Our investment into distributed data structures means that Loki could one day support multi-GPU architectures.

9 CONCLUSION

This paper presents our new multi-physics framework, Loki, as a generalized tool for robust simulation of various phenomena at state-of-the-art fidelity in the VFX industry. Distinguished from previous simulation tools in production, Loki's coupling is considered key from the very beginning of its design and is hence on by default for many types of elements, including elastic solids, rigid bodies, fluids, and secondaries. The front-end configuration is carefully planned so that users do not have to understand or define the order of execution in the solver, despite creating very complex multiple-element setups with regular feature updates. The resulting system can deal with scenes from many millions of elements on a single machine to multi-billion elements distributed over clusters. With all these advantages, Loki has successfully been used in many major productions since 2018 and has seen steadily increasing artist adoption even when given the option of any major third-party solver.

ACKNOWLEDGMENTS

We are thankful to the many people besides the authors of this article who contributed to Loki. Niall Ryan and Michael Forot were particularly influential in laying a strong early architectural foundation. John Farrow, Nikolay Ilinov, Marcus Schoo, Greg Klar, Dan Elliott, Stephen Ward, Clemens Sielaff, Vincent Bonnet, Marie-Lena Eckert, Andreas Soderstrom, Pavel Jurkas, and Andrew Harvey each contributed multiple years of research or engineering development to Loki. We are also thankful for leadership and management support from Christoph Sprenger, Ken Museth, Millie Maier, Risha Patel, Barton Gawboy, Sam Martin, Julia Jones, Daniel Hodson, Luca Fascione, and Joe Marks.

Many talented artists have worked closely with the Simulation Research team and were critical for guiding Loki to be successful through their constructive feedback and long-term championing of the system. Thank you to Gary Boyle, Kevin Blom, Johnathan Nixon, Alex Nowotny, David Caeiro, Nicholas Illingworth, Adrien

Rollet, Christopher Dean, Rahul Deshprabhu, Florian Hu, and Ziggy Kucas from the FX department. Thank you to Gios Johnston, John Homer, Julian Butler, Andrea Merlo, Tim Teramoto, Jefri Haryono, and Carlos Lin from the Creatures department. And thank you to Louis-Daniel Poulin, Alex Klaricich, and Stephen Cullingford from additional artist departments.

Thank you to Tamar Shinar and Jon Hertzig who provided feedback and editing on this article, and thank you to Kayvon Fatahalian whose "What Makes a (Graphics) Systems Paper Beautiful" notes gave us the structure needed to undertake writing a systems paper. Finally, Loki was made possible thanks to Joe Letteri whose vision for better visual effects through fundamental understanding of the physical world gave us the time, resources, and invaluable guidance to keep us true to that path.

10 GLOSSARY

Action A developer class dedicated to modifying simulation data stored in one or more Objects. Derived classes include Forces, Constraints, and Operators. 7

Behavior A user-facing construct that defines a high-level unit of computation such as adding an Object into the scene or operating on the existing Objects. 4

Bucket A container that stores spatially divided simulation data. Buckets may be iterated over in parallel, or transferred between ranks in an MPI-enabled simulation. 8

Constraint An Action that calculates non-negative energies from groups of degrees of freedom, and calculates the contributions to the system of equations to minimize this energy. An example is spring attachments. 10

Force An Action that calculates a force to be applied to an Object, with the assumption that the force calculation does not depend on neighboring degrees of freedom. Examples include gravity and fluid drag. 10

Grid Domain A partitioning of space where Buckets are allocated sparsely for grid cells containing simulation data. 8

Group A user-facing construct that may contain Behaviors or Groups recursively. All Behaviors within the Group interact with one another. 4

Object A representation of degrees of freedom that is simulated forward in time and may store a mix of internal data representations. Examples of Objects include Curves Object for hair simulation or Volume Object for wind simulation. 7

Operator An Action that affects Objects in ways that do not fit into the paradigm of Constraints or Forces. An example is vorticity confinement. 12

Solver Setup A user-facing collection of Groups, Behaviors, and parameters that define how simulation data is read and processed. 4

Stage A discrete, statically defined period of time when Behaviors can schedule execution of specific Tasks. 6

Task A unit of computation required by a Behavior to be performed at a specific Stage. 5

REFERENCES

Muzaffer Akbay, Nicholas Nobles, Victor Zordan, and Tamar Shinar. 2018. An Extended Partitioned Method for Conservative Solid-Fluid Coupling. *ACM Trans. Graph.* 37, 4, Article 86 (jul 2018), 12 pages.

Ryoichi Ando and Christopher Batty. 2020. A Practical Octree Liquid Simulator with Adaptive Surface Resolution. *ACM Trans. Graph.* 39, 4, Article 32 (jul 2020), 17 pages.

R. Barrett, M.W. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics. https://books.google.co.kr/books?id=8IkWgiZ8kOwC

Christopher Batty, Florence Bertails, and Robert Bridson. 2007. A Fast Variational Framework for Accurate Solid-Fluid Coupling. In *ACM SIGGRAPH 2007 Papers (SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 100–es.

Jan Bender, Kenny Erleben, and Jeff Trinkle. 2014. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Comput. Graph. Forum* 33, 1 (feb 2014), 246–270.

Miklós Bergou, Basile Audoly, Etienne Vouga, Max Wardetzky, and Eitan Grinspun. 2010. Discrete Viscous Threads. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 116, 10 pages.

Miklós Bergou, Max Wardetzky, Stephen Robinson, Basile Audoly, and Eitan Grinspun. 2008. Discrete Elastic Rods. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 63, 12 pages.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. 35, 2, Article 21 (may 2016), 12 pages. https://doi.org/10.1145/2892632

Morten Bojsen-Hansen, Michael Bang Nielsen, Konstantinos Stamatelos, and Robert Bridson. 2021. Spatially Adaptive Volume Tools in Bifrost. In *ACM SIGGRAPH 2021 Talks (SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 2, 2 pages. https://doi.org/10.1145/3450623.3464642

Eddy Boxerman and Uri Ascher. 2004. Decomposing Cloth. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '04)*. Eurographics Association, Goslar, DEU, 153–161. https://doi.org/10.1145/1028523.1028543

Christopher Brandt, Leonardo Scandolo, Elmar Eisemann, and Klaus Hildebrandt. 2019. The Reduced Immersed Method for Real-Time Fluid-Elastic Solid Interaction and Contact Simulation. *ACM Trans. Graph.* 38, 6, Article 191 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356496

George E. Brown, Matthew Overby, Zahra Forootaninia, and Rahul Narain. 2018. Accurate Dissipative Forces in Optimization Integrators. *ACM Trans. Graph.* 37, 6, Article 282 (dec 2018), 14 pages.

Gilles Daviet. 2020. Simple and Scalable Frictional Contacts for Thin Nodal Objects. *ACM Trans. Graph.* 39, 4, Article 61 (jul 2020), 16 pages.

Gilles Daviet and Florence Bertails-Descoubes. 2016. A Semi-Implicit Material Point Method for the Continuum Simulation of Granular Materials. *ACM Trans. Graph.* 35, 4, Article 102 (jul 2016), 13 pages.

Gilles Daviet, Florence Bertails-Descoubes, and Laurence Boissieux. 2011. A Hybrid Iterative Solver for Robustly Capturing Coulomb Friction in Hair Dynamics. *ACM Trans. Graph.* 30, 6 (dec 2011), 1–12.

Denis Demidov. 2020. AMGCL - A C++ library for efficient solution of large sparse linear systems. *Software Impacts* 6 (2020), 100037.

Pradeep Dubey, Pat Hanrahan, Ronald Fedkiw, Michael Lentine, and Craig Schroeder. 2011. PhysBAM: Physically Based Simulation. In *ACM SIGGRAPH 2011 Courses (SIGGRAPH '11)*. Association for Computing Machinery, New York, NY, USA, Article 10, 22 pages.

David Eberle. 2018. Better Collisions and Faster Cloth for Pixar's Coco. In *ACM SIGGRAPH 2018 Talks (SIGGRAPH '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 2 pages. https://doi.org/10.1145/3214745.3214801

François Faure, Christian Duriez, Hervé Delingette, Jérémie Allard, Benjamin Gilles, Stéphanie Marchesseau, Hugo Talbot, Hadrien Courtecuisse, Guillaume Bousquet, Igor Peterlik, and Stéphane Cotin. 2012. SOFA: A Multi-Model Framework for Interactive Physical Simulation. In *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*, Yohan Payan (Ed.). Studies in Mechanobiology, Tissue Engineering and Biomaterials, Vol. 11. Springer, 283–321.

Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. 2001. Visual Simulation of Smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 15–22.

Yun (Raymond) Fei, Christopher Batty, Eitan Grinspun, and Changxi Zheng. 2018. A Multi-Scale Model for Simulating Liquid-Fabric Interactions. *ACM Trans. Graph.* 37, 4, Article 51 (jul 2018), 16 pages.

Yun (Raymond) Fei, Henrique Teles Maia, Christopher Batty, Changxi Zheng, and Eitan Grinspun. 2017. A Multi-Scale Model for Simulating Liquid-Hair Interactions. *ACM Trans. Graph.* 36, 4, Article 56 (jul 2017), 17 pages.

Gene H. Golub and Charles F. van Loan. 2013. *Matrix Computations* (fourth ed.). JHU Press. http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm

Eitan Grinspun, Anil N Hirani, Mathieu Desbrun, and Peter Schröder. 2003. Discrete shells. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Citeseer, 62–67.

Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. 2005. Coupling Water and Smoke to Thin Deformable and Rigid Shells. In *ACM SIGGRAPH 2005 Papers (SIGGRAPH '05)*. Association for Computing Machinery, New York, NY, USA, 973–981.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

Ales Janka. 2007. Smoothed aggregation multigrid for incompressible flows. *PAMM* 7 (12 2007), 1025901 – 1025902.

Chenfanfu Jiang, Theodore Gast, and Joseph Teran. 2017. Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact. *ACM Trans. Graph.* 36, 4, Article 152 (jul 2017), 14 pages.

Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. 2016. The Material Point Method for Simulating Continuum Materials. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. Association for Computing Machinery, New York, NY, USA, Article 24, 52 pages.

Richard Jones and Richard Southern. 2017. Physically-Based Droplet Interaction. In *Proceedings of the ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA '17)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages.

Theodore Kim and David Eberle. 2020. Dynamic Deformables: Implementation and Production Practicalities. In *ACM SIGGRAPH 2020 Courses (SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 182 pages. https://doi.org/10.1145/3388769.3407490

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (mar 2016), 21 pages. https://doi.org/10.1145/2866569

Gergely Klár, Theodore Gast, Andre Pradhana, Chuyuan Fu, Craig Schroeder, Chenfanfu Jiang, and Joseph Teran. 2016. Drucker-Prager Elastoplasticity for Sand Animation. *ACM Trans. Graph.* 35, 4, Article 103 (jul 2016), 12 pages.

Frank Losasso, Tamar Shinar, Andrew Selle, and Ronald Fedkiw. 2006. Multiple Interacting Liquids. *ACM Trans. Graph.* 25, 3 (jul 2006), 812–819.

Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. 2008. Two-Way Coupled SPH and Particle Level Set Fluid Simulation. *IEEE Transactions on Visualization and Computer Graphics* 14, 4 (2008), 797–804. https://doi.org/10.1109/TVCG.2008.37

Chaoyang Lyu, Wei Li, Mathieu Desbrun, and Xiaopei Liu. 2021. Fast and Versatile Fluid-Solid Coupling for Turbulent Flow Simulation. *ACM Trans. Graph.* 40, 6, Article 201 (dec 2021), 18 pages. https://doi.org/10.1145/3478513.3480493

Miles Macklin and Matthias Müller. 2013. Position Based Fluids. *ACM Trans. Graph.* 32, 4, Article 104 (jul 2013), 12 pages.

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: Position-Based Simulation of Compliant Constrained Dynamics. In *Proceedings of the 9th International Conference on Motion in Games (MIG '16)*. Association for Computing Machinery, New York, NY, USA, 49–54.

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. 2019. Small Steps in Physics Simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '19)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages.

Sebastian Martin, Peter Kaufmann, Mario Botsch, Eitan Grinspun, and Markus Gross. 2010. Unified Simulation of Elastic Rods, Shells, and Solids. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 39, 10 pages.

A. McAdams, E. Sifakis, and J. Teran. 2010. A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '10)*. Eurographics Association, Goslar, DEU, 65–74.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Trans. Graph.* 30, 4, Article 37 (jul 2011), 12 pages.

Eder Miguel, Rasmus Tamstorf, Derek Bradley, Sara C. Schvartzman, Bernhard Thomaszewski, Bernd Bickel, Wojciech Matusik, Steve Marschner, and Miguel A. Otaduy. 2013. Modeling and Estimation of Internal Friction in Cloth. *ACM Trans. Graph.* 32, 6, Article 212 (nov 2013), 10 pages.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position Based Dynamics. *J. Vis. Comun. Image Represent.* 18, 2 (apr 2007), 109–118.

Ken Museth. 2013. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.* 32, 3, Article 27 (jul 2013), 22 pages. https://doi.org/10.1145/2487228.2487235

Ken Museth. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation *(SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 2 pages. https://doi.org/10.1145/3450623.3464653

Rahul Narain, Jonas Zehnder, and Bernhard Thomaszewski. 2019. A Second-Order Advection-Reflection Solver. *Proc. ACM Comput. Graph. Interact. Tech.* 2, 2, Article 16 (jul 2019), 14 pages.

Saket Patkar, Mridul Aanjaneya, Dmitriy Karpman, and Ronald Fedkiw. 2013. A Hybrid Lagrangian-Eulerian Formulation for Bubble Generation and Dynamics. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '13)*. Association for Computing Machinery, New York, NY, USA, 105–114.

Avi Robinson-Mosher, Tamar Shinar, Jon Gretarsson, Jonathan Su, and Ronald Fedkiw. 2008. Two-Way Coupling of Fluids to Rigid and Deformable Solids and Shells. *ACM Trans. Graph.* 27, 3 (aug 2008), 1–9.

Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An Unconditionally Stable MacCormack Method. *J. Sci. Comput.* 35 (06 2008), 350–371.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation. *ACM Trans. Graph.* 33, 6, Article 205 (nov 2014), 12 pages. https://doi.org/10.1145/2661229.2661269

Morten Silcowitz, Sarah Niebe, and Kenny Erleben. 2010. Projected Gauss-Seidel Subspace Minimization Method for Interactive Rigid Body Dynamics - Improving Animation Quality using a Projected Gauss-Seidel Subspace Minimization Method. *GRAPP 2010 - Proceedings of the International Conference on Computer Graphics Theory and Applications* 229, 38–45.

Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable Neo-Hookean Flesh Simulation. *ACM Trans. Graph.* 37, 2, Article 12 (mar 2018), 15 pages.

Jos Stam. 1999. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., USA, 121–128.

Jos Stam. 2009. Nucleus: Towards a unified dynamics solver for computer graphics. In *2009 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*. 1–11. https://doi.org/10.1109/CADCG.2009.5246818

Alexey Stomakhin, Andrew Moffat, and Gary Boyle. 2019. A Practical Guide to Thin Film and Drips Simulation. In *ACM SIGGRAPH 2019 Talks (SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 72, 2 pages.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A Material Point Method for Snow Simulation. *ACM Trans. Graph.* 32, 4, Article 102 (jul 2013), 10 pages.

Alexey Stomakhin, Craig Schroeder, Chenfanfu Jiang, Lawrence Chai, Joseph Teran, and Andrew Selle. 2014. Augmented MPM for Phase-Change and Varied Materials. *ACM Trans. Graph.* 33, 4, Article 138 (jul 2014), 11 pages.

Alexey Stomakhin, Joel Wretborn, Kevin Blom, and Gilles Daviet. 2020. Underwater Bubbles and Coupling. In *ACM SIGGRAPH 2020 Talks (SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 2 pages.

Tetsuya Takahashi and Christopher Batty. 2020. Monolith: A Monolithic Pressure-Viscosity-Contact Solver for Strong Two-Way Rigid-Rigid Rigid-Fluid Coupling. *ACM Trans. Graph.* 39, 6, Article 182 (nov 2020), 16 pages.

Tetsuya Takahashi and Christopher Batty. 2021. FrictionalMonolith: A Monolithic Optimization-Based Approach for Granular Flow with Contact-Aware Rigid-Body Coupling. *ACM Trans. Graph.* 40, 6, Article 206 (dec 2021), 20 pages.

Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. 2015. Smoothed Aggregation Multigrid for Cloth Simulation. *ACM Trans. Graph.* 34, 6, Article 245 (oct 2015), 13 pages.

Alessandro Tasora, Radu Serban, Hammad Mazhar, Arman Pazouki, Daniel Melanz, Jonathan Fleischmann, Michael Taylor, Hiroyuki Sugiyama, and Dan Negrut. 2016. Chrono: An Open Source Multi-physics Dynamics Engine. In *Lecture Notes in Computer Science*. Springer International Publishing, 19–49.

Yun Teng, David I. W. Levin, and Theodore Kim. 2016. Eulerian Solid-Fluid Coupling. *ACM Trans. Graph.* 35, 6, Article 200 (nov 2016), 8 pages. https://doi.org/10.1145/2980179.2980229

Maxime Tournier, Matthieu Nesme, Benjamin Gilles, and François Faure. 2015. Stable Constrained Dynamics. *ACM Trans. Graph.* 34, 4, Article 132 (jul 2015), 10 pages.

Pascal Volino, Nadia Magnenat-Thalmann, and Francois Faure. 2009. A simple approach to nonlinear tensile stiffness for accurate cloth simulation. *ACM Transactions on Graphics* 28, 4, Article 105 (2009).

Joel Wretborn, Sean Flynn, and Alexey Stomakhin. 2022. Guided Bubbles and Wet Foam for Realistic Whitewater Simulation. *ACM Trans. Graph.* 41, 4, Article 117 (jul 2022). https://doi.org/10.1145/3528223.3530059

Tao Yang, Jian Chang, Bo Ren, Ming C. Lin, Jian Jun Zhang, and Shi-Min Hu. 2015. Fast Multiple-Fluid Simulation Using Helmholtz Free Energy. *ACM Trans. Graph.* 34, 6, Article 201 (oct 2015), 11 pages. https://doi.org/10.1145/2816795.2818117

David M. Young. 1971. Chapter 4 - CONVERGENCE OF THE BASIC ITERATIVE METHODS. In *Iterative Solution of Large Linear Systems*, David M. Young (Ed.). Academic Press, 106–139.

Yongning Zhu and Robert Bridson. 2005. Animating Sand as a Fluid. *ACM Trans. Graph.* 24, 3 (jul 2005), 965–972.