# CSIG-2023 Project Report

## *A High-Performance and Comprehensive Computation Framework for Real-Time SPH Fluid Simulation, Visualization and Interaction with Cloth*

SPHere Group

*Zhu Zihang, Lu Xinyu, Luo Xukun, Lu Zixian, Huang Kemeng*

Pre
September 9, 2023

## SPHere = SPH + Here = sphere

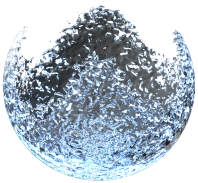


*Figure:* SPH fluid with sphere bounding, remeshed by Houdini and rendered by LuisaRender

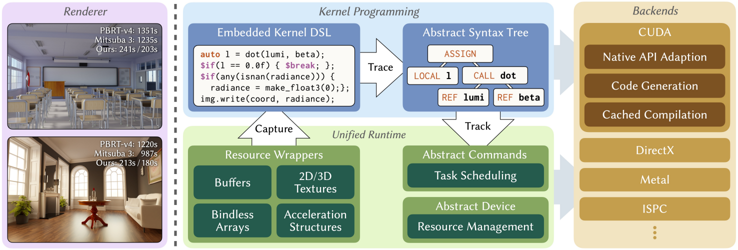
1. Lu Xinyu, TransGP(HKU)
2. Zhu Zihang, NJU
3. Luo Xukun, ISCAS
4. Lu Zixian, ISCAS
5. Huang Kemeng, HKU

## TOC

## Feature List

- **Modular Design**: Modular design and package management system **SPHerePackage**
- **Physics Simulation**:
  - SPH (PCISPH, WCSPH)
  - Task-based general real-time particle neighbor search algorithm
  - General hardware ray tracing collision detection
  - XPBD cloth simulation
  - XPBD cloth-SPH water weak coupling
  - SDF static boundary interaction
- **Rendering and Visualization**: Real-time camera, real-time fluid cloth rendering
- **Parallel Primitives**: Reduce/Scan
- **File IO**: abc/obj/sdf/smesh
- **Asset Generation**: cloth builder/fluid builder
- **User Interface**: QT-GUI
- **Build System**: Xmake

## LuisaCompute



```
auto l = dot(lumi, beta);
$if(l == 0.0f) { $break; };
$if(any(isnan(radiance))) {
  radiance = make_float3(0);};
img.write(coord, radiance);
```

*Figure:* LuisaCompute Architecture [5]

## Functions in LC

1. **Kernel**: The entry function of device.
2. **Callable**: The function that can be called by Kernel.
3. **Inline Function**: The function that can be called by Kernel, but will be inlined in Kernel.

## Capture

- **Variable Capture**: Capture the value of host variable. For basic types, the value is the runtime value of the variable. For complex types, the value is the runtime value of the pointer.
- **Resource Capture**: Capture the handle of host resource. LC will pass the handle of the resource to device implicitly.

## Multi-stage Code Generation

1. C++ Preprocessor
2. C++ Template
3. C++ Runtime AST Generation
4. AST to Device Code Generation

   *Stage 1 and 2 are common C++ code generation stages. Stage 3 and 4 are unique to LC. With LC's **C++ embedded DSL** (C++ Embedded Domain Specific Language), SPHerePackage could increse its ability of code reuse and modularity.*

## Just In Time Compilation

1. Runtime Compilation
2. Cut-off Branches
3. Optimize Register Allocation
4. Reduce Memory Barrier without decrease code reuse

## Command Reordering

1. Stream Agnostic
2. Reduce Synchronization
3. Let User Decide When to Execute

## LC helps the optimize your code on the backend

These principles are quoted from *https://gpuopen.com/learn/rdna-performance-guide/*

1. assure the number of allocators greator than the thread recoding
2. try to reuse the same allocator
3. minimize the Amount of Command Buffer Submission to GPU
4. use pipeline cache to reuse PSO
5. create static large memory heaps and sub-allocated from the heap
6. ...

## Package Environment

- **PackageGlobal** a thread local static class, is the package environment context of the current thread.
- **PackageEnvScope** a Guard class, whose construction and destruction determine a specific package environment. When constructed, push the current environment to PackageGlobal, and when destructed, restore the original environment of PackageGlobal. If PackageEnvScope is not used, it will always be in the default environment.
- **Package** base class, used to standardize behavior encapsulation and provide auxiliary code. All package developments inherit from this base class.

## Package Type

1. **Method Package**: Provide Callable for users. It does not need to be compiled and only has AST Gen stage.
2. **Routine Package**: Provide Kernel for users. Users provide input data that meets the requirements and call the package to get the result (such as the parallel primitive library implemented in this project). This kind of package needs to be compiled before it can be used.
3. **Module Package**: Provide Kernel/Callable for users. Different from 1/2, this kind of package maintains a private data structure (occupying device resources). Users interact with the algorithm of this package through the interface provided by this package. Typical Module Packages in SPHere are: BVHCollisionDetection (Kernel with resources), SDF (Callable with resources).
4. **Inline Package**: Provide inline code segment for users. Users directly call the corresponding C++ function, and the corresponding code is expanded in Callable or Kernel, used for helper functions.

## Package Norm

- **config**: Each package must have a config stage, which is used to determine the capture behavior of the package code in the AST Gen stage.
- **astgen**: Callable, Kernel can generally be completed by lazy generation.
- **compile**: For Routine Package and Module Package, there is a compile stage, which can be compiled by lazy compilation (compile when the user calls a specific interface).

## Package Management

- Callable Package, Routine Package can be managed and reused, Module Package cannot be reused, because the latter has specific resources.
- The package management of **SPHerePackage** maintains a Description, Package. Any package needs to be uniquely described by Description. The package that meets the description will be returned to the user when PackageManager::require(Description) is called. If there is no package that meets the description, the corresponding package will be created and returned to the user.
- Packages can depend on each other, and this dependency is also implemented through require

## Callable Package

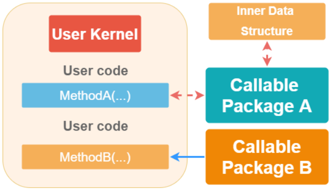


*Figure:* Callable Package

## *Callable Package*

- Callable Package can be divided into two categories:
  1. **Module** with resources, that is, this kind of package needs to apply for resources such as Buffer, Image from device, and use it as the data structure inside the package. Users can access the data inside the package through the interface provided by the package. Corresponding to Package A in Figure 3.
  2. **Method** without resources, that is, this kind of package does not need to apply for any resources from device. All functions provided by this package operate on the resources provided by the user, or are just helper functions. Corresponding to Package B in Figure 3.
- For Callable Package, users will first configure and astgen the package, and then use it in their Kernel. As shown in Figure 3:

## Kernel Package

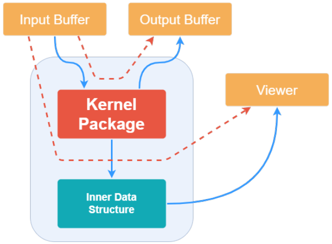


*Figure:* Module Package

## Kernel Package

Kernel Package can be divided into two categories:

1. **Module** with resources, that is, this kind of package needs to apply for resources such as Buffer, Image from device, and use it as the data structure inside the package. Corresponding to Input Buffer→Viewer path in Figure 4.
2. **Routine** without resources, that is, this kind of package does not need to apply for any resources from device. All resources are applied and provided by the user. Corresponding to Input Buffer→Output Buffer path in Figure 4.

## SPH Method

*SPH(Smoothed-particle hydrodynamics) is a space discrete method, which mainly estimates the value of continuous function in space by sampling points around it. For example, the value of continuous function $A(x)$ at position $x$ is calculated by the physical quantity of particles in the neighborhood of the position, and the interpolation is calculated by the smooth kernel function $W$. The specific kernel function interpolation formula is as follows:*

$$A(x) = \sum_j m_j \frac{A_j}{\rho_j} W(x - x_j, h), \tag{1}$$

## SPH Method

$$A(x) = \sum_j m_j \frac{A_j}{\rho_j} W(x - x_j, h)$$

1. $m_j$ is the mass of the particle.
2. $\rho_j$ is the density of the particle.
3. h is the radius of the smooth kernel (support domain).
4. W is the smooth kernel function, whose influence should decrease with the increase of distance. The kernel function W selected in this project is cubic spline function.

$$W(q) = \sigma_d \begin{cases} 6(q^3 - q^2) + 1, & \text{for } 0 \leq \text{q} < 0.5 \\ 2(1-q)^3, & \text{for } 0.5 \leq \text{q} < 1 \\ 0, & \text{for q} \geq 1 \end{cases} \tag{2}$$

## N-S Equation

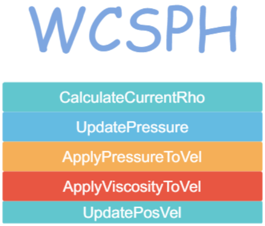
*using SPH method to calculate the simplified Naiver-Stokes equation. Among them, the NS equation is divided into momentum equation and continuity equation:*

$$\begin{aligned} \frac{d\rho}{dt} &= -\rho\nabla \boldsymbol{v} \quad \text{(continuity equation)} \\ \frac{d\boldsymbol{v}}{dt} &= -\frac{1}{\rho}\nabla P + \mu\nabla^2\boldsymbol{v} + \boldsymbol{g} \quad \text{(momentum equation)} \end{aligned} \tag{3}$$

## SPH Method to Solve NS Equation

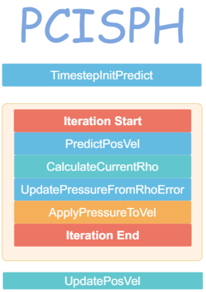
1. **WCSPH** (Weakly Compressibility SPH) is a SPH method based on state equation. This method assumes that the fluid can be slightly compressed and calculates the pressure by the compression amount.
2. **PCISPH** (Predictive-Corrective Incompressible SPH) is a SPH method based on iterative correction. This method assumes that the fluid is incompressible and iteratively corrects the pressure to maintain incompressibility.

## WCSPH



1. assume that the fluid can be slightly compressed, calculate the pressure by the compression amount.
2. calculate current density $\rho(\boldsymbol{x}) = \sum_b m_b W(\boldsymbol{x} - \boldsymbol{x_b}, h)$
3. calculate pressure using Tait Equation: $P(\boldsymbol{x}) = B((\frac{\rho(\boldsymbol{x})}{\rho_0})^\gamma - 1)$
4. when neighborhood pressure is inconsistent, particles will be pushed away or pulled together. $\nabla f_p(\boldsymbol{x}) = -\frac{\nabla P(\boldsymbol{x})}{\rho(\boldsymbol{x})}$.
5. calculate the viscosity force, using artificial viscosity as *SPlisHSPlasH*

## PCISPH

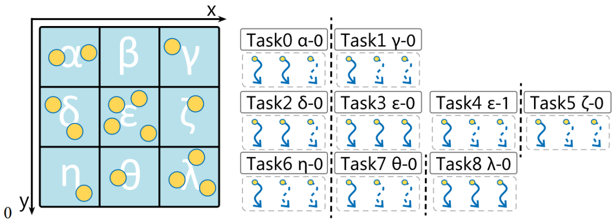


PCISPH is based on WCSPH, with some slight changes:

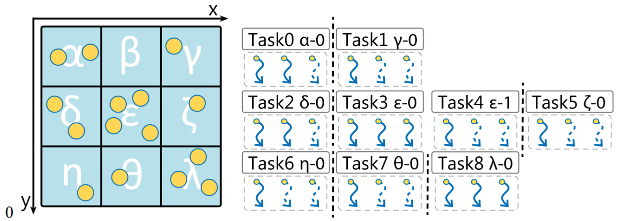
1. **Prediction Step** predicte the position and velocity of the particle.
2. **Correction Step** calcuate the difference of predictive density and static density, and correct the pressure.
3. recursively correct the pressure until the density error is less than a certain threshold [3]

## Neighbor Search

*We implement the task-based neighborhood search method [1]*

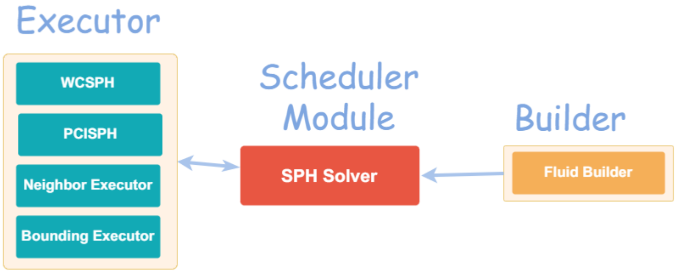


1. split the space into a uniform grid, which width, length and height are all support radius.
2. assign each particle to the grid cell it belongs to.



1. assign particles in grid cell to the task of length 32, maintaining all the particles in the same task are in the same grid cell.
2. assign each task to 32 thread, when the distribution of particles are sparse, the number of threads will be less than 32.
3. for sparse case: each particle is assigned to a thread, and the neighborhood data is obtained and the physical quantity is calculated.
4. for dense case: each task shares the neighborhood data.

## SPH Solver Design



## XPBD Solver

XPBD is a simple deformable simulation method, which is:

- fast(matrix-free),
- easy to implement,
- constraint-based ,
- stable.

can be used to simulate:

- cloth : mass-spring, strain based ...
- hair/rod: rod dynamics ...
- soft body: strain based ...
- rigid body: shape matching ...
- fluid

XPBD starts with Newton's equation of motion given by:

$$\mathbf{M}\ddot{\mathbf{x}} = -\nabla U(\mathbf{x})^T + \mathbf{f}_{ext}, \tag{4}$$

| | |
|---|---|
| $\mathbf{M}$ | the lumped mass matrix |
| $\mathbf{x}$ | $\left(\mathbf{x}_0{}^T, \mathbf{x}_1{}^T, \mathbf{x}_2{}^T, \ldots\right)^T$ |
| $U$ | the elastic potential |
| $\mathbf{f}_{ext}$ | the stacked external force vector |

Apply the implicit Euler discretization to Eq. 4:

$$\mathbf{M}\frac{\mathbf{x} - \hat{\mathbf{x}}}{\Delta t^2} = -\nabla U(\mathbf{x})^T, \tag{5}$$

where $\mathbf{x} := \mathbf{x}(t + \Delta t)$ and $\hat{\mathbf{x}} = 2 \cdot \mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \mathbf{M}^{-1} \cdot \mathbf{f}_{ext} \cdot \Delta t^2$.

All potentials in the XPBD framework are formulated as squared constraints:

$$U(\mathbf{x}) = \frac{1}{2}\mathbf{C}(\mathbf{x})^T \alpha^{-1}\mathbf{C}(\mathbf{x}),$$

where $\alpha$ is the diagonal compliance matrix describing the inverse stiffness of all constraints.

By taking the negative gradient of the potential term, we obtain the elastic force:

$$\mathbf{f}_{\text{elastic}} = -\nabla U^T = -\nabla \mathbf{C}^T \alpha^{-1}\mathbf{C} = \frac{1}{\Delta t^2}\nabla \mathbf{C}^T \lambda,$$

where $\lambda = -\tilde{\alpha}^{-1}\mathbf{C}(\mathbf{x})$ with $\tilde{\alpha} = \frac{\alpha}{\Delta t^2}$.

Thus, we obtain two equations available for iteration:

$$\mathbf{M}(\mathbf{x}-\tilde{\mathbf{x}})-\nabla\mathbf{C}(\mathbf{x})^T\lambda=\mathbf{0}, \tag{6}$$

$$\mathbf{C}(\mathbf{x})+\tilde{\alpha}\lambda=\mathbf{0}. \tag{7}$$

We rewrite Eqs. 6 and 7 as $\mathbf{g}(\mathbf{x},\lambda)=\mathbf{0}$ and $\mathbf{h}(\mathbf{x},\lambda)=\mathbf{0}$, respectively, and obtain the following equation by linearizing Eqs. 6 and 7:

$$\begin{bmatrix} \mathbf{K} & -\nabla\mathbf{C}^T(\mathbf{x}^i) \\ \nabla\mathbf{C}(\mathbf{x}^i) & \tilde{\alpha} \end{bmatrix}\begin{bmatrix} \Delta\mathbf{x} \\ \Delta\lambda \end{bmatrix}=-\begin{bmatrix} \mathbf{g}\left(\mathbf{x}^i,\lambda^i\right) \\ \mathbf{h}\left(\mathbf{x}^i,\lambda^i\right) \end{bmatrix}, \tag{8}$$

where $\mathbf{x}^i$ denotes the result of the $i$-th iteration, and

$$\mathbf{K}=\frac{\partial\mathbf{g}}{\partial\mathbf{x}}=\mathbf{M}-\frac{\partial^2 C^T}{\partial\mathbf{x}^2}\cdot\lambda,$$

By plugging the approximations of XPBD, $\mathbf{K} \approx \mathbf{M}$ and $\mathbf{g} \approx \mathbf{0}$, into Eq. 8, we get the following equations to describe the iteration process of XPBD:

$$\left[\nabla \mathbf{C}\left(\mathbf{x}^i\right) \mathbf{M}^{-1} \nabla \mathbf{C}\left(\mathbf{x}^i\right)^T + \tilde{\alpha}\right] \Delta\lambda = -\mathbf{C}\left(\mathbf{x}^i\right) - \tilde{\alpha}\lambda^i, \tag{9}$$

$$\Delta \mathbf{x} = \mathbf{M}^{-1} \nabla \mathbf{C}\left(\mathbf{x}^i\right)^T \Delta\lambda. \tag{10}$$

Since $\mathbf{M}$ and $\tilde{\alpha}$ are diagonal matrices, Eqs. 9 and 10 can be expressed in a matrix-free style. Using $j$ to represent the $j$-th constraint, and $k$ to represent the index of the particle affected by the $j$-th constraint, we have

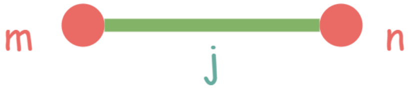
$$\Delta\lambda_j = \frac{-C_j - \tilde{\alpha}_j \lambda_j^i}{\nabla C_j \mathbf{M}_j^{-1} \nabla C_j^T + \tilde{\alpha}_j}, \tag{11}$$

$$\Delta\mathbf{x}_k = m_k^{-1} \frac{\partial C_j}{\partial \mathbf{x}_k}^T \Delta\lambda_j, k \in \mathscr{O}_j, \tag{12}$$

where $\mathscr{O}_j$ is a set containing the indices of all particles affected by the $j$-th constraint. $\mathbf{M}_j$ is the lumped mass matrix of all particles in $\mathscr{O}_j$. For instance, if the $j$-th length constraint expresses the length relationship between particle $m$ and particle $n$, then $\mathscr{O}_j = \{m, n\}$.

## XPBD Spring Example

$$C(\mathbf{x}_m, \mathbf{x}_n) = \|\mathbf{x}_m - \mathbf{x}_n\|$$



## Our Constraints

- Length Constraint $C(\mathbf{x}_0, \mathbf{x}_1) = \|\mathbf{x}_0 - \mathbf{x}_1\| - l_0 = 0$
- Triangle Strain Constraint $C(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) = \det(\mathbf{F}) - 1 = 0.$
- External Collision Constraint $C(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) = -(d(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) - d_{\text{thickness}}) \leq 0.$
- Fluid Cloth Collision Constraint $C(\mathbf{x}_{\text{fluid}}, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) = -(d(\mathbf{x}_{\text{fluid}}, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2) - d_{\text{thickness}} - r_{\text{fluid}}) \leq 0.$

## Solver Implementation

- **XPBD Solver**: Core Solver
  - Constraint Solver:Solve all kinds of constraints
  - Integrator:Integrate particles in time
  - Particle Manager:Manage particle attributes, manage the results of various constraint solutions and apply a specific normalization strategy
  - Topology Manager:Maintain the topology of particles, such as recording specific information on edges, triangles, and quadrilaterals.
- **Builder**: Tools for generating solvable objects from assets
  - ClothBuilder:Cloth asset generation, generate basic grid cloth (Grid), generate cloth from .obj file, etc.
  - ClothPatch:Representation of cloth patches, used to initialize cloth, download current cloth solution results from GPU, etc.
  - ClothMaterial Cloth material.

## Solver Implementation

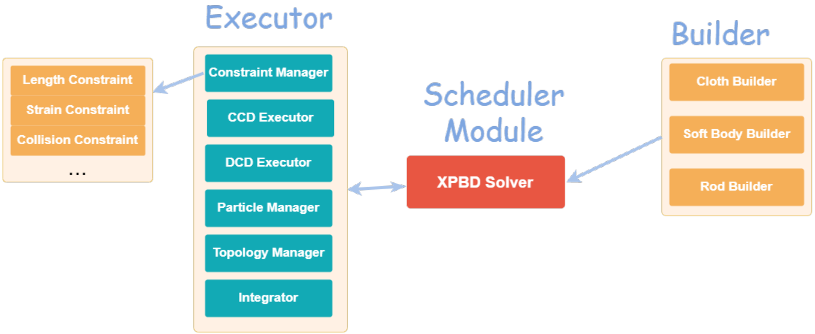


*Figure:* XPBD Solver

## Coupling

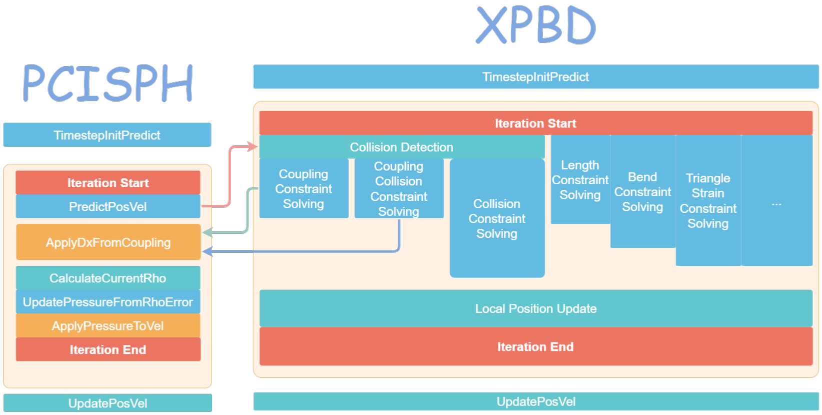


*Figure:* SPH-XPBD Coupling

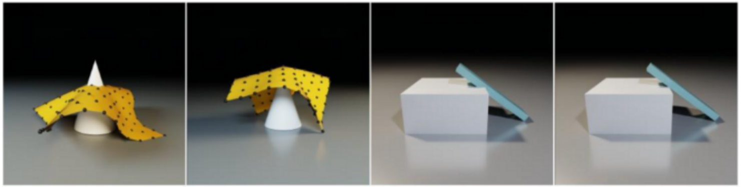
1. SPH Predict Phase: Update the position of SPH particles
2. XPBD Collision Detection Phase: Wait for the completion of SPH Predict Phase
3. SPH Second Update Phase: Wait for the completion of XPBD Collision Detection Phase
4. After that, SPH and XPBD solvers can work independently until the iteration ends.

## Coupling with External Collider

To coupling SPH and XPBD with external collider, we employ the SDF-based collision method from [2], using Frank-Wolfe algorithm to solve the following non-linear constrained optimization problem:

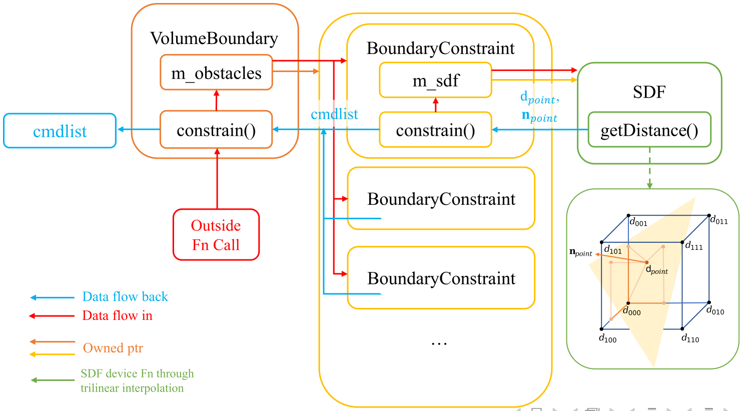
$$\mathbf{x}_i = \arg\min_{\alpha,\beta,\gamma} \phi(\mathbf{x}_i),$$

$$s.t.\ \mathbf{x}_i = \alpha\mathbf{p}_i + \beta\mathbf{q}_i + \gamma\mathbf{r}_i, \alpha+\beta+\gamma = 1.$$



## Our Implementation

The following figure shows the data flow and function calls of the **constrain()** function when the fluid interacts with the static SDF boundary module.



## Ray Tracing Collision Detection

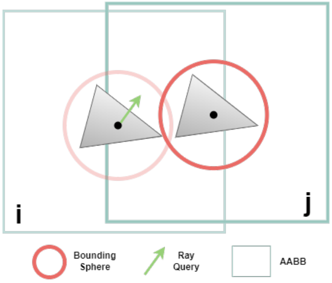
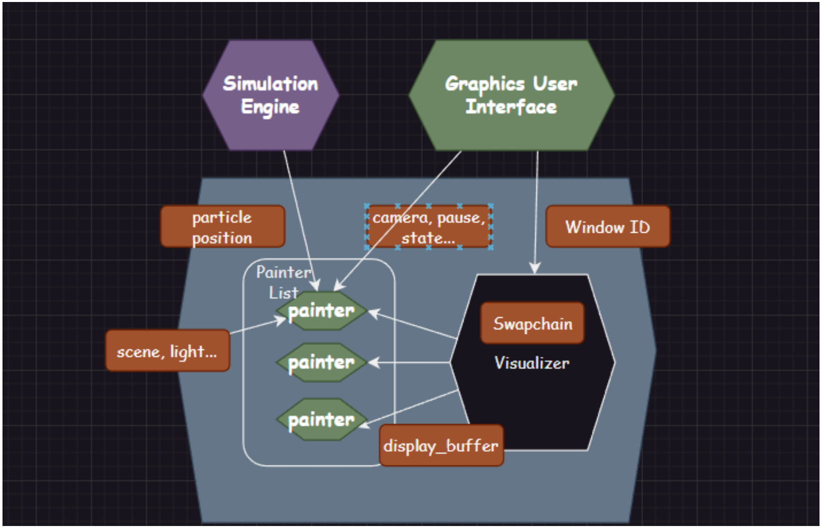


*Figure:* Ray Tracing Collision Detection

The core function of hardware ray tracing is BVH acceleration structure construction, Ray-AABB and Ray-Triangle detection. We use BVH acceleration structure construction and Ray-AABB part to realize the collision detection algorithm. [4]

## Visualization Framework



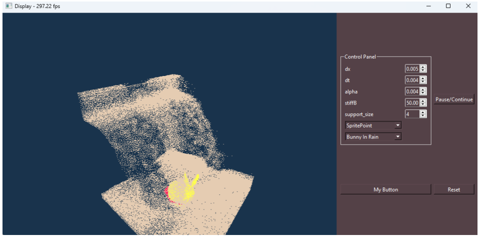
## Visualization Framework

- **GUI** is responsible for displaying the simulation screen externally, accepting and processing user input and converting it into configuration information, and controlling the entire update process.
- The **Visualizer** internally maintains a swapchain, maintains the painter registered as a painter list, and calls the paint method of each painter in turn to render the image on the display buffer during the rendering update, and finally synchronizes the display buffer to the foreground.
- The **Painter** is responsible for the specific visualization algorithm, obtains the particle position information from the simulation engine, obtains the scene information such as the light source from the scene configuration, and obtains the camera position and direction update information from the GUI.

## Visualization Method

- **Rasterization**: use point to draw fluid, use mesh for cloth, SDF, etc.
- **Screen Space Fluid**: use screen space depth and thickness buffer to reconstruct fluid surface
- **Volume Rendering**: use ray marching to render fluid

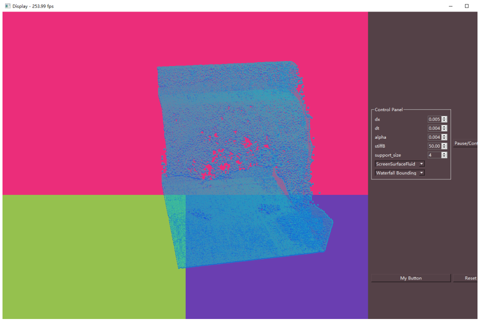
## Rasterization



dx 0.005
dt 0.004
alpha 0.004
stiffB 50.00
support_size 4
SpritePoint
Bunny In Rain

- Advantage: can support multiple object
- Drawback: expensive to reconstruct fluid surface

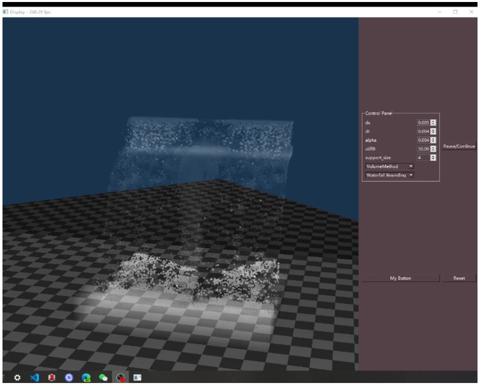
## Screen Space Fluid



Control Panel
dx 0.005
dt 0.004
alpha 0.004
stiffB 50.00
support_size 4
ScreenSurfaceFluid
Waterfall Bounding

- Advantage: fast, easy to get result
- Drawback: cannot be extended to more complex geometry of fluid, light and shadow rendering, the reconstructed normal is not accurate.

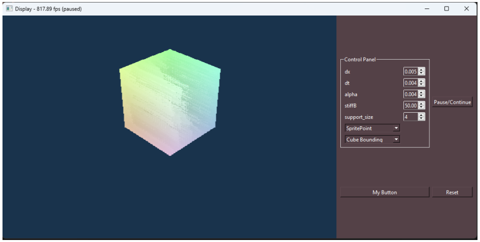
## *Volume Rendering*



dx 0.005
dt 0.004
alpha 0.004
stiffB 50.00
support_size 4
VolumeMethod
Waterfall Bounding

- Advantage: accurate, can be extended to more complex geometry of fluid, light and shadow rendering
- Drawback: expensive to render

## GUI Method



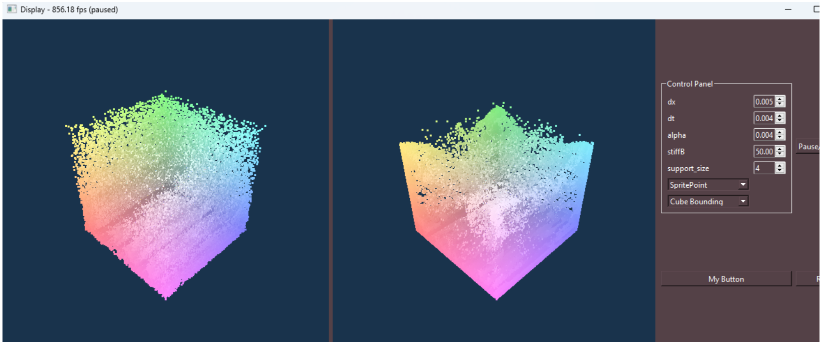
- MainWindowQT: main window inherited from QMainWindow
- Main widget inherited from QWidget
  1. Canvas: visualization part
  2. Control Panel: control part

## Control Flow

*Based on QT's slot and signal, we can build our control flow on our simulation engine.*

- **Simulation Process Control**: start/pause/reset
- **Camera Control**: rotate/zoom/pan/move
- **Parameter Control**: change parameters like dx, dt, alpha, stiffB, etc.
- **Scene Control**: change scene like waterfall, cloth, etc.
- **Visualization Control**: change visualization method

## Comparison View



Display - 856.18 fps (paused)

Control Panel

| | |
|---|---|
| dx | 0.005 |
| dt | 0.004 |
| alpha | 0.004 |
| stiffB | 50.00 |
| support_size | 4 |

SpritePoint

Cube Bounding

## Benchmark

- **Cube**: the competition scene, one 0.5m x 0.5m x 0.5m cube of water falls into a 1m x 1m x 1m cube container, with 125,000 particles, 1000 static density, 30% energy absorption rate, -0.7 reflection coefficient, 0.02 search radius.
- **Sphere**: change the container to a sphere, two render our logo.
- **Waterfall**: a waterfall scene, show the flexibility of our scene modification ablitity.
- **Bunny**: a bunny with a cloak under the waterfall, show the coupling of fluid, cloth and static boundary.

## Performance

The Result Was Tested on RX7900 GRE

| Scene | Cube(SP) | Cube(SSF) | Cube(Volume) | Bvh | Sphere | Waterfall |
|---|---|---|---|---|---|---|
| FPS | 691 | 522 | 446 | 72 | 471 | 702 |

- **SP** means the scene is rendered by SpritePoint (a kind of points splatting method), the default visualization method.
- **SSF** means the scene is rendered by ScreenSpaceFluid (a kind of screen space method).
- **Volume** means the scene is rendered by Volumetric Rendering Method (with screen-space intermediate area reconstruction).

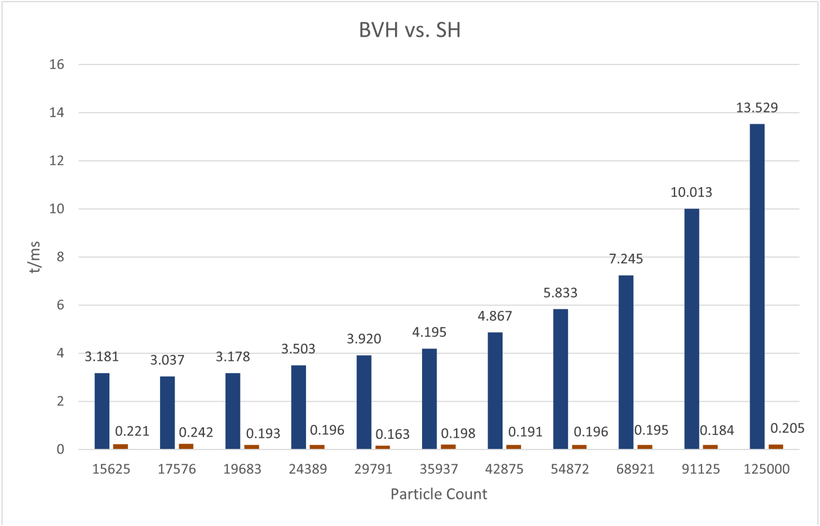
## Performance

*Software Spatial Hash performs better than Hardware BVH?*

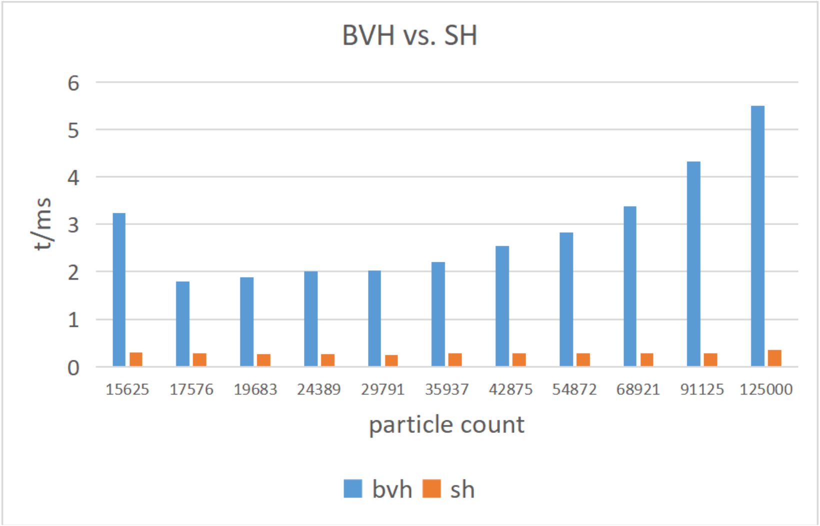
| fps | BVH | Spatial Hash |
|---|---|---|
| RTX4090 | 175 | 630 |
| RX7900 | 72 | 691 |

*As shown in Figure 9, the performance of Spatial Hash is about ten times that of BVH. When SPH simulation, the particles are often very dense locally, which leads to the BVH nodes being traversed frequently (Ray AABB detection), which in turn leads to the performance degradation of BVH. It can be said that BVH is more suitable for culling than for neighborhood search.*

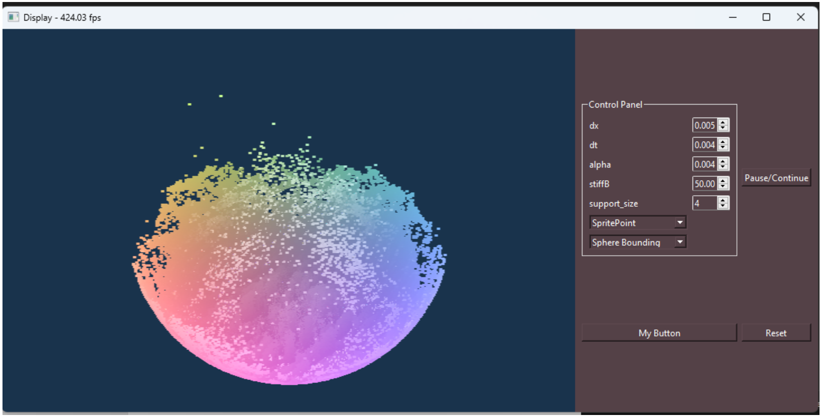
# *BVH and Spatial Hash Profiling on RX7900*



## BVH and Spatial Hash Profiling on RTX4090



## SPHere Logo


Display - 424.03 fps

Control Panel

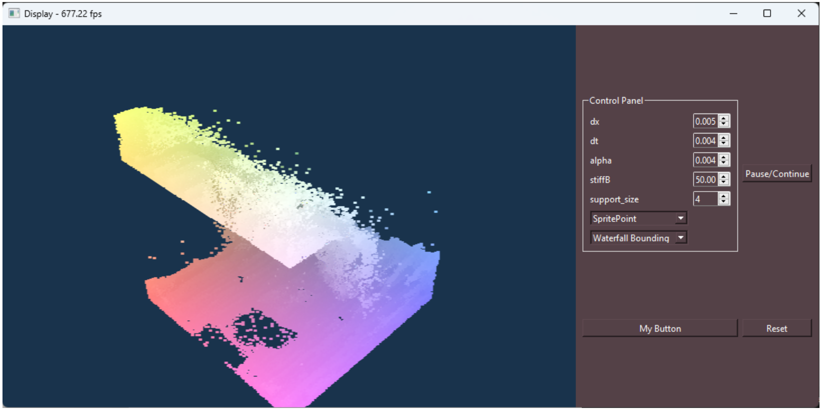
| | |
|---|---|
| dx | 0.005 |
| dt | 0.004 |
| alpha | 0.004 |
| stiffB | 50.00 |
| support_size | 4 |

SpritePoint

Sphere Bounding

*Figure:* SPHere Logo

## Waterfall


Display - 677.22 fps

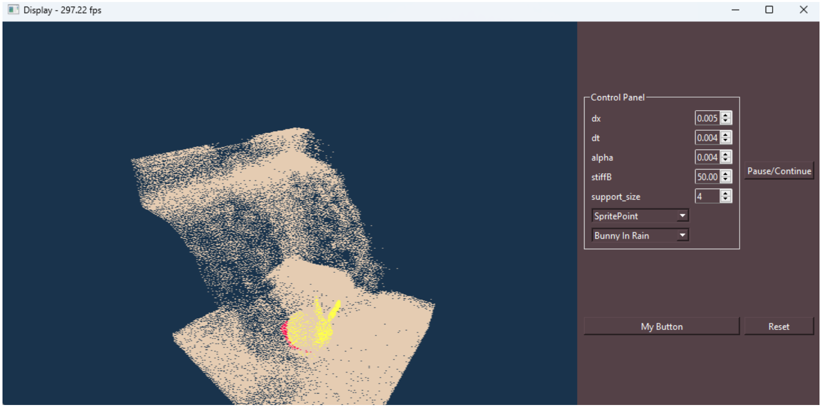
Control Panel

| | |
|---|---|
| dx | 0.005 |
| dt | 0.004 |
| alpha | 0.004 |
| stiffB | 50.00 |
| support_size | 4 |

SpritePoint
Waterfall Bounding

*Figure:* Waterfall

## Bunny in Rain


Display - 297.22 fps

Control Panel
dx 0.005
dt 0.004
alpha 0.004
stiffB 50.00
support_size 4
SpritePoint
Bunny In Rain

*Figure:* Bunny in Rain

## Reference I

[1] Kemeng Huang, Jiming Ruan, Zipeng Zhao, Chen Li, Changbo Wang, and Hong Qin.
A general novel parallel framework for sph-centric algorithms.
*Proc. ACM Comput. Graph. Interact. Tech.*, 2(1), jun 2019.

[2] Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Zach Corse.
Local optimization for robust signed distance field collision.
3(1):1–17, 2020.

[3] Barbara Solenthaler and Renato Pajarola.
Predictive-corrective incompressible sph.
In *ACM SIGGRAPH 2009 papers*, pages 1–6. 2009.

## Reference II

[4] Shiwei Zhao, Zhengshou Lai, and Jidong Zhao.
Leveraging ray tracing cores for particle-based simulations on gpus.
*International Journal for Numerical Methods in Engineering*, 10 2022.

[5] Shaokun Zheng, Zhiqian Zhou, Xin Chen, Difei Yan, Chuyan Zhang, Yuefeng Geng, Yan Gu, and Kun Xu.
Luisarender: A high-performance rendering framework with layered and unified interfaces on stream architectures.
*ACM Trans. Graph.*, 41(6), nov 2022.