

# 不可压流体的模拟、渲染和应用研究

## 摘要

计算机图形学中，流体的实时模拟和渲染在游戏、工程分析、灾害预警等领域具有广泛的应用。然而，当前普遍使用的拉式算法（Lagrangian method）普遍具有真实性差的缺陷：模拟方面，算法为换取更好的性能，常舍弃流体的不可压性质，导致模拟结果偏离物理真实；渲染方面，算法很少对模拟粒子做平滑处理，使得渲染结果表现出不真实的粒子效果。

针对流体的实时模拟和渲染问题，本文使用 CUDA 并行框架和 OpenGL GLSL 语言，给出了一套基于位置的流体（Position Based Fluids）模拟与屏幕空间渲染算法的并行实现。模拟方面，本文使用雅可比迭代并行求解密度平衡方程，修正流体的不可压性，并综合考虑了流体的表面张力和粘着力，给出了稳定、一致的无偏模拟结果。渲染方面，本文在屏幕空间液体渲染的框架下，提出使用双边滤波作为液体深度纹理平滑的方法。该方法能很好地保持液体的边缘，避免了边缘处理不当造成的层次感缺失。同时，本文综合了菲涅尔方程，比尔-朗博定律和光线追踪方法，给出了具有真实感的液体表面着色效果。实验结果显示，该方案兼顾了实时性与真实性，能够在严苛的实时应用场景下，提供真实可信的液体模拟和渲染结果。

**关键词：**计算机图形学，拉式模拟，OpenGL，屏幕空间渲染，不可压流体，实时应用，CUDA，GPU 并行计算

装

订

线

# The Simulation, Rendering and Application of Incompressible Fluids

## ABSTRACT

In Computer Graphics, real-time simulation and rendering of fluids is widely used in Gaming, Engineering Analysis, Disaster Alerting and other fields. However, current widely adopted algorithms are deficient in realism: In terms of simulation, these algorithms tend to omit incompressibility of the fluids for better real-time performance, resulting in a divergence from physical reality; In terms of rendering, they usually lack proper smoothing for the fluids surface and render them in a undesired blobby or jelly-like style.

To address the challenge of fluids simulation and rendering, we introduce a parallel implementation of Position Based Fluids simulation and Screen-Space Fluids rendering using CUDA computing platform and OpenGL Shading Language. For simulation, we use Jacobi iteration to solve density equilibrium equation in order to adjust incompressibility over the time. We incorporate Jacobi with surface tension and viscosity refinement to synthesis a stable, consistent and unbiased simulation result. For rendering, we suggest using bilateral filter as depth smoothing technique under the framework of screen-space rendering. Bilateral filter can properly preserve the silhouette and avoid the loss of depth level. Our method leverages Fresnel's law, Beer-Lambert's law and ray-tracing to give a persuasive shading of fluid surface. Our experiments show that our method can produce authentic result even under strict real-time requirement.

**Key words:** computer graphics, Lagrangian simulation, OpenGL, screen-space rendering, incompressible fluids, real-time application, CUDA, Parallel Computing on GPU

订

线

## 目 录

1 引言 .....	1
1.1 研究背景 .....	1
1.2 流体模拟的研究现状与相关工作 .....	1
1.3 流体渲染的研究现状与相关工作 .....	2
1.4 本文所作的工作 .....	3
1.5 本文结构安排 .....	3
2 流体的 PBF 模拟方法 .....	4
2.1 Navier-Stokes 方程组 .....	4
2.1.1 动量方程 .....	4
2.1.2 不可压条件 .....	7
2.2 平滑粒子动力学模型 .....	8
2.2.1 平滑核 .....	8
2.2.2 SPH 流体参数 .....	9
2.2.3 SPH 流体模拟框架 .....	11
2.2.4 SPH 流体的特征和缺陷 .....	12
2.3 求解不可压性的 PBF 方法 .....	12
2.3.1 流体位置的压强修正 .....	13
2.3.2 流体速度的表面张力修正 .....	15
2.3.3 流体的粘度修正 .....	16
2.4 PBF 方法的 GPU 并行化实现 .....	17
2.4.1 实现管线概述 .....	17
2.4.2 CUDA 并行计算概述 .....	19
2.4.3 CUDA 核函数和数据结构设计 .....	21
2.4.4 哈希网格邻居寻找算法 .....	22
2.4.5 实现细节 .....	24
3 拉式流体的屏幕空间渲染 .....	28
3.1 算法流程概述 .....	28
3.2 渲染管线的坐标变换 .....	30
3.3 表面法向量重建 .....	34
3.4 深度平滑 .....	36
3.5 表面着色 .....	38
4 结果与讨论 .....	42
4.1 结果展示 .....	42
4.1.1 单立方液体 .....	42
4.1.2 双立方液体 .....	42
4.1.3 边缘移动下的液体 .....	44
4.2 性能分析 .....	45
4.2.1 粒子数量性能分析 .....	45
4.2.2 模拟阶段性能分析 .....	46
5 结论和展望 .....	48
5.1 结论 .....	48
5.2 展望 .....	48

---

5.2.1 模拟部分 .....	48
5.2.2 渲染部分 .....	48
参考文献 .....	50
谢 辞 .....	53

装

订

线

## 1 引言

### 1.1 研究背景

本文研究具有真实感的流体实时模拟和渲染问题。计算流体力学（Computational Fluid Dynamics, CFD）是使用数值方法模拟流体运动的学科，流体模拟对象包括水流、火焰、烟雾、流动的沙粒等一系列自然现象和景观。航空航天、传播制造等领域对模拟的物理真实性有很高要求。与其不同，计算机图形学中常常牺牲一定的物理准确性，以较小的计算量，追求视觉上的真实性。计算机游戏等交互性较强的应用，更是对流体的模拟和渲染提出了严格的实时性要求。由于传统的模拟方法计算量庞大，并行化程度低，不能很好满足实时性要求，流体模拟的实时化逐渐成为业界和学界研究的热点。

### 1.2 流体模拟的研究现状与相关工作

根据微分方程离散方式的不同，模拟方法通常可分为欧式法（Euler method）、拉式法（Lagrangian method）和混合方法（Hybrid method）三类。本世纪初 Stam 等人发表的两篇重要论文[1][2]，成为了欧式法的奠基之作。欧式法将空间分割为离散的格子（Grid），将流体的速度、压强、温度等物理参数记录在格子中。通过将 Navier-Stokes（NS）方程进行有限差分，欧式法将微分方程求解转化为高阶线性方程的求解。针对欧式法均匀采样浪费空间，损失精度的问题，Losassao 提出了八叉树[3]，Feldman[4]等人提出了非结构四面体网格（Unstructured tetrahedral meshes）的离散方法。然而，这些方法的实现十分复杂，同时精度通常逊于朴素的欧式方法。传统欧式法依旧存在过高内存要求、采样精度损失和数值耗散等问题[5]。2016 年，Chern 等人发表了论文《薛定谔的烟》[6]，提出使用 $C^2$ 为值域的波函数描述流体状态，并使用凝聚物理中用于求解超流体的非线性薛定谔方程（Gross-Pitaevskii 方程）作为流体的动力学方程，开发了一种没有数值耗散，同时可以高效求解其不可压性的欧式方法。这种方法在对烟雾的模拟上取得了突破性的效果。但对于需要渲染表面的水流等液体，目前并没有好的波函数表面重建方法。

拉式法将流体离散为固定数量，在约束下自由活动的粒子，由粒子记录流体状态的各个物理量。拉式法可追溯到 20 世纪 70 年代 Gingold、Lucy 等人在研究星际气体运动的工作[7][8]。Desbrun 和 Cani[9]最先把 Gingold 的平滑粒子流体动力学方法（Smoothed Particle Hydrodynamics, SPH）引入图形学中。随后，Müller 的工作[10]展现了 SPH 在实时动画和交互应用中的前景。Solenthaler 等人提出的预测校正不可压 SPH（Predictive-corrective incompressible, SPH）[11]第一次实现了满足不可压条件的拉式方法，将拉式方法推进了一大步，并在实时模拟领域得到了广泛的应用[12][13][14]，但其需要将一帧分割为较小的步长的多步，以保持算法的稳定性。随后

Macklin 等人提出基于位置的流体 (Position Based Fluids) [15]，该算法能够在与 PCISPH 同等精度的条件下，以较大的模拟步长得出稳定的模拟结果，具有较好的实时应用前景[16]。

混合方法是目前电影工业界常用的，精度最高的方法。混合方法使用欧式法求解不可压方程，欧式法精度高且易于处理流体边界；使用拉式法处理流体的对流 (Advection)，即各物理量随着流体在速度场的转移，从而避免了欧式法的数值耗散。经典的混合方法包括八十年代流行至今的粒子元胞法 (Particle-In-Cell, PIC) [17]，和隐式粒子流体法 (FLuid Implicit Particle, FLIP) [18]。商业软件如 Houdini, Realflow 几乎都是使用 FLIP 方法。最近 Jiang 等人提出了仿射粒子元胞法 (Affine Particle-In-Cell, APIC) [19]，其兼具了 PIC 的稳定性和 FLIP 低耗散的特点。其改进，Fu 提出的多项式粒子元胞法 (A polynomial particle-in-cell method) [20]，大幅减少了能量和旋量 (vorticity) 在数值计算中的损失。APIC 被已经应用在迪士尼和皮克斯最新的电影中（如《海洋奇缘》）[21]。

以上三种典型模拟方法中，拉式模拟具有比其他两者低得多的时间和空间复杂度。由于现有的硬件条件的限制，拉式模拟是当前实时流体模拟应用唯一实际的选择。其中，拉式方法中的 PBF 方法具有步长大下稳定性好的优点，是目前拉式模拟的最新成果。因此本文模拟部分将基于 PBF 方法展开对其并行实现的研究。

### 1.3 流体渲染的研究现状与相关工作

几乎所有的流体渲染的工作都涉及流体表面的重建。根据重建方法的不同，流体渲染方法大致可以分为隐式表面重建 (Implicit Surface Construction)，显式 (Explicit) 表面重建和屏幕空间 (Screen-space) 渲染三类。液体表面重建完毕之后，便可使用同在多边形网格 (Mesh) 渲染中使用的各类着色模型。因此，本节重点回顾和评估各类表面重建方法中的重要文献。

隐式表面重建算法是最古老、文献最多也是应用最广的算法。这类算法的开山之作移动立方体法 (Marching Cubes) 由 Lorensen 和 Cline 发表于 SIGGRAPH 1987[22]。原始的移动立方体法存在匹配形态的歧义，Nielson 和 Hamann 提出使用渐进线判决器 (Asymptotic decider) 解决了这个问题[23]。近十年来，Rosenberg 和 Birdwell 特别针对粒子等值面提取的问题优化了立方体匹配方法，使其能够做到在 3K 的粒子的负荷下实时运行。但 SPH 算法中为了真实性效果常常使用 10K 以上的粒子。另一个算法，Williams 提出的贴片匹配 (Marching Tiles) 能够实现很好的表面平滑效果，但同样存在实时性问题[24]。显示表面重建如[25]常常涉及复杂的网格操作，并且难以并行化，同样不适用于实时应用。

较为理论的屏幕空间渲染概念可追溯到[26]。Müller 在这篇文章中提出了一种从深度纹理还原可视表面多边形网格的方法。该方法计算代价高昂，且难以在 GPU 上实现。两年后，[24]提出了从深度纹理直接渲染液体表面的屏幕空间液体渲染方法。该方法与 Müller 同样使用了平滑的深度纹理，然后直接计算得到液体表面法向量以进行着色。[24]使用了表面曲率的概念作为表面平滑的幅度因子。算法本质上相当于应用了一个 $2 \times 2$  大小的卷积核，通过将这个卷积核在深度

装

订

线

纹理上应用多次以达到平滑的效果。实际上，这个卷积核的构造代价相当高昂。本文在[24]框架上应用了不同的平滑算法，以更高效率得到了更优的结果。

## 1.4 本文所作的工作

本文针对不可压流体的模拟和渲染，分别给出了基于位置的流体（Position based fluids）方法[15]和屏幕空间液体渲染（Screen-Space Fluid Rendering）的并行实现。本文的贡献包括以下几点：

- (1) 以实时性为目标，本文使用 CUDA 并行框架和 OpenGL GLSL 语言作为并行计算工具，给出了一套 PBF 模拟与屏幕空间渲染算法的 GPU 并行实现。
- (2) 本文提出在[24] 的框架上使用双边滤波作为液体深度纹理平滑的方法。实验表明，该方法在还原平滑的液体表面的同时，很好的保持了液体的边缘，避免了边缘处理不当造成的层次感缺失。
- (3) 本文综合了菲涅尔方程，比尔-朗博定律和光线追踪算法，提出了一套具有真实感的液体表面着色模型。

## 1.5 本文结构安排

第 1 章：本章介绍了计算机图形学中流体模拟和渲染的研究背景和研究意义，回顾了文献中流体模拟和渲染的主要流派和方法，叙述了本文的主要工作和结构安排。

第 2 章：本章首先叙述了描述流体运动的 Navier-Stokes 方程和它的 SPH 离散方法，然后详细介绍了本文使用的 PBF 方法及其 GPU 并行化实现。

第 3 章：本章介绍了本文使用的屏幕空间液体渲染算法，包括 OpenGL 空间变换，液体深度纹理、厚度纹理的渲染，表面法向量还原，深度纹理平滑和液体表面着色等部分。

第 4 章：本章首先展示了本文实现的模拟渲染算法在几个典型测试场景下的表现，然后评估了算法的性能表现。

第 5 章：本章对本文所做工作进行总结，并提出未来可能的研究方向。

订

线

## 2 流体的 PBF 模拟方法

一切计算机物理模拟都是对描述物理现象的物理方程的有限近似和数值求解。本节首先从流体的物理性质出发，导出准确描述流体运动的 Navier-Stokes 方程，然后介绍了本文使用的空间离散方法——平滑粒子动力学（Smoothed Particle Hydrodynamics, SPH）模型。平滑粒子动力学模型是拉式模拟的一种典型实现方法，但它本身不描述流体的运动规律。本节接着引入了 Navier-Stokes 方程的时间离散方法——“基于位置的流体 (Position Based Fluids, PBF)” 方法。这种方法最突出贡献，是使用雅可比方法迭代修正不可压方程，从而以很小的代价、优越的强健性解决了流体不可压约束的问题。本节最后描述了使用 CUDA 的 PBF 方法并行实现。

### 2.1 Navier-Stokes 方程组

#### 2.1.1 动量方程

Navier-Stokes 方程组是描述流体运动规律的方程组。本文研究的流体满足不可压条件和牛顿条件两个条件。后文将详细描述两个条件和它们的使用范围。从这两个条件出发，本节导出 Navier-Stokes 方程组的完整形式。

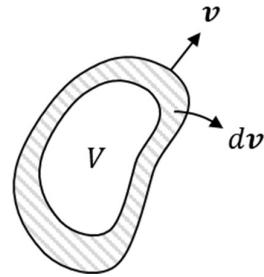


图 2.1 体积  $V(t)$  随时间  $t$  变化，变化量微元为  $dV$

装

首先本文考虑空间中的一块流体区域  $V$ ，这块区域内的任意一点有密度  $\rho$ 、速度  $\mathbf{v}$  等物理量。

设外力为  $\mathbf{F}$ ，对这一块流体区域应用牛顿第二定律，有

$$\frac{d}{dt} \int_{V(t)} \rho \mathbf{v} dV = \mathbf{F} \quad (2.1)$$

订

注意 (2.1) 中研究的对象是一块固定的流体，含有固定的一些分子。由于流体会流动，区域  $V(t)$  是随时间变化的量。(2.1) 式的左侧可以用莱布尼茨法则展开。莱布尼茨法能够将对积分的导数转化成对导数的积分，如下所示， $f$  是液体的任意一个物理量， $S$  是区域  $V$  的边界，有

$$\frac{d}{dt} \int_{V(t)} f dV = \int_{V(t)} \frac{df}{dt} dV + \int_{\partial V(t)} f dS \quad (2.2)$$

线

$$= \int_{V(t)} \frac{df}{dt} dV + \int_{S(t)} f \mathbf{v} \cdot \mathbf{n} dS$$

其中方程右侧  $\int_{dV(t)} f dV = \int_{S(t)} f \mathbf{v} \cdot \mathbf{n} dS$  是应用了三维空间中的散度定理（高斯定律）。

莱布尼茨法则有一个显著的物理意义。方程左侧物理量  $f$  在整块区域的变化，是右侧第一项区域内  $f$  随时间变化，与右侧第二项体积变化叠加的结果。

将莱布尼茨法则应用到 (2.1) 左侧，可以得到

$$\frac{d}{dt} \int_{V(t)} \rho \mathbf{v} dV = \int_{V(t)} \left[ \mathbf{v} \left( \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) \right) + \rho \left( \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) \right] dV \quad (2.3)$$

上式比较复杂，但实际上，本文可以利用质量守恒定律做简化。空间  $V$  虽然会随时间发生变动，但本文考虑的其中的流体既不会增加也不会减少，因此有**质量守恒定律**

$$\frac{d}{dt} \int_{V(t)} \rho dV = 0 \quad (2.4)$$

对质量守恒定律应用莱布尼茨法则，有

$$\frac{d}{dt} \int_{V(t)} \rho dV = \int_{V(t)} \left[ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) \right] dV = 0$$

因为对任意的  $V$  上面的等式都成立，必有  $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$ ，而这一项恰好出现在 (2.3) 中，因此可将 (2.3) 简化。将简化的 (2.3) 带回 (2.1)，得到牛顿定律导出的最终方程，文献中常称之为**动量方程**。

$$\int_{V(t)} \rho \left( \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) dV = \mathbf{F} \quad (2.5)$$

如果将 (2.5) 与牛顿第二定律方程的基本形式  $m\mathbf{a} = \mathbf{F}$  相比较，能够发现两者之间形式上的相似性： $\rho$  对应  $m$ ，同时  $\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}$  中的  $\frac{\partial \mathbf{v}}{\partial t}$  正好是加速度的定义，而多出来的一项  $\mathbf{v} \cdot \nabla \mathbf{v}$  是“漂移”“产生加速度”。为了便于理解，考虑一个具体的例子：假设有一条河流，河面上每个位置的水流速度是固定的，但是不同位置的速度是不同的。当一个人泛舟而下时，假设船速与水流速度相等，他将会感受到船的加速度，这个加速度是因为船经过了速度不同的区域。 $\mathbf{v} \cdot \nabla \mathbf{v}$  中  $\nabla \mathbf{v}$  表征了速度变化的方向，而  $\mathbf{v} \cdot \nabla \mathbf{v}$  表征了漂移的方向。

为了简化表示，定义材料微分运算符  $\frac{D}{Dt}$

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla \quad (2.6)$$

利用 (2.6) 将 (2.5) 重写为 (2.7)

$$\int_{V(t)} \rho \frac{D\mathbf{v}}{Dt} dV = \mathbf{F} \quad (2.7)$$

现在考虑作用在体积  $V$  上的外力  $\mathbf{F}$ , 它们的存在使得液体的速度发生改变。首先容易想到,  $\mathbf{F}$  应包含有重力  $\mathbf{F}_g = \int_V \mathbf{g} \rho dV$ , 其中  $\mathbf{g}$  是重力常数。除此之外, 流体会还受到周围流体的挤压, 产生压强力  $\mathbf{F}_{gp}$ 。对于有粘性的流体, 还会因为与周围流体有流速差异受到粘性力  $\mathbf{F}_v$ 。下面本文将逐项分析。

首先考虑压强力  $\mathbf{F}_p$ 。压强仅在体积  $V$  的表面  $S$  对  $\mathbf{F}_p$  有贡献, 因为内部的压强力将会互相抵消。对于流体, 通常可以认为压强是各向同性的, 即一点处的压强对四周任意方向施加的压强力大小是相同的, 因此压强是一个标量。

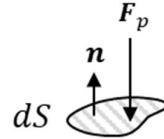


图 2.2 片元压强

对于表面  $S$  上的任意一块小片元  $dS$ , 设此处压强为  $p$ , 表面法向量为  $\mathbf{n}$ , 由压强的定义容易知道  $d\mathbf{F}_p = p\mathbf{n}dS$ 。因此作用在整块体积上的压强力为

$$\begin{aligned} \mathbf{F}_p &= \int_S -p\mathbf{n}dS \\ &= \int_V -\nabla p dS \end{aligned} \quad (2.8)$$

装

上式中的第二个等号应用了散度定理。

最后本文来考虑粘性力  $\mathbf{F}_v$ 。理想的无粘性、不可压的液体被称为欧拉液体, 又称为超流体。这种液体没有粘性力, 在没有外力扰动的情况下, 它的动能和旋度将始终保持不变——其中的液体漩涡一旦生成, 永远也不会消失。超流体是真实存在的, 已知的超流体包括超低温状态下的氦-3 和氦-4 液体[27]。在天体物理和高能物理中也有关于超流状态的物质的理论[28]。

最常见的流体, 例如水和烟尘, 都可被认为属于牛顿流体。通俗的讲, 牛顿流体的特征是其粘性力正比于流速的变化速率。如图 2.3 所示, 在流体的一个断层两端有不同的流速, 则快的一端将会对慢的一端施加拖拽力。牛顿流体的这种性质使得不同位置的速度趋于相等, 同时液体的能量发生损耗, 最终达到静态平衡状态。

订

线

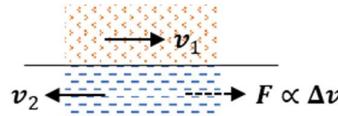


图 2.3 断层上的粘性力

流体速度的梯度  $\nabla \mathbf{v}$  表征空间不同方向上流速变化的强度和方向。在区域边界  $S$  上任意一处片元  $dS$ , 本文考虑粘性力  $d\mathbf{F}_v$  对体积  $V$  的作用。对垂直边界法向方向上的速度分量, 根据牛顿第三定律, 它受到的粘性力必有一等大反向的粘性力作用在另一片元上。而由于此方向上的速度必定是沿着平行  $dS$  方向变化, 此反作用力没有离开表面  $S$ , 因此此力与其反作用力对体积  $V$  没有影响。由此推理, 对体积  $V$  有影响的是切向流向的速度在法向方向变化产生的粘性力, 因此此力正比于  $\mathbf{n} \cdot \nabla \mathbf{v}$ 。应当注意到, 在三维空间中,  $\nabla \mathbf{v}$  是一个  $3 \times 3$  张量,  $\mathbf{n} \cdot \nabla \mathbf{v}$  则得到一个 3 维向量。在表面上对这个式子积分时, 散度定理依然适用。因篇幅原因此处不做赘述。

在表面  $S$  上对粘性力积分就得到了作用在整个体积  $V$  上的粘性力, 由如下公式给出

$$\mathbf{F}_v = \int_S \mu \mathbf{n} \cdot \nabla \mathbf{v} dV = \int_V \mu \nabla^2 \mathbf{v} dV \quad (2.9)$$

其中  $\mu$  是粘度因子, 或称动态粘度 (Dynamic viscosity)。通常假设牛顿液体的动态粘度是固定的。在实际情况下, 动态粘度会随着压强、温度甚至磁场发生变化。例如, 低温状态下的蜂蜜比高温状态下的蜂蜜拥有大得多的粘度。然而, 这不是本文研究的重点, 因此本文假设一种流体的粘度始终是一个常数。

综合重力  $\mathbf{F}_g$ , 压强力  $\mathbf{F}_p$  和粘性力  $\mathbf{F}_v$ , 带入动量方程 (2.5) 中, 本文导出完整的动量方程的积分形式和微分形式。模拟算法是作用在离散模型上的算法, 更贴合方程的微分形式。本文将在 2.3 节中展开利用动量方程微分形式更行平滑粒子状态的方法。

$$\int_{V(t)} \rho \left( \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) dV = \int_V \mathbf{g} \rho dV + \int_V -\nabla p dS + \int_V \mu \nabla^2 \mathbf{v} dV \quad (2.10)$$

$$\rho \frac{D\mathbf{v}}{Dt} = \rho \mathbf{g} - \nabla p + \mu \nabla^2 \mathbf{v} \quad (2.11)$$

### 2.1.2 不可压条件

在密度  $\rho$ , 重力常数  $\mathbf{g}$ , 粘度  $\mu$  都给定的情况下, 动量方程 (2.11) 仍并不足以给出流体的一个解, 因为流体的压强  $p$  是一个随时间和位置变化的变量。向量方程 (2.11) 在  $x, y, z$  上给出三个标量方程, 但速度  $\mathbf{v}$  和压强  $p$  构成了 4 个未知量。因此要使方程确定必须要寻找一个额外条件, 这个额外条件恰好是不可压条件。

$$\nabla \cdot \mathbf{v} = 0 \quad (2.12)$$

动量方程 (2.11) 和不可压条件 (2.12) 统称为 Navier-Stokes 方程组。

不可压条件是密度为常数和物质守恒的必然结果。不可压条件的物理意义是，对一块本文确定的体积 $V$ ，其中含有固定的流体分子，无论体积 $V$ 的形状怎么变化，它的大小都不会发生改变。考虑一段时间内体积 $V$ 的变化量与其表面处液体流速的关系

$$0 = dV = \int_S \mathbf{n} \cdot \mathbf{v} dS = \int_V \nabla \cdot \mathbf{v} dS \quad (2.13)$$

第一个等号是因为质量守恒 $dm = \rho dV = 0$ ，第二个等号参看图 2.1。三个等号应用了散度定理。因为对任意 $V$ 上式都成立，所以有不可压条件 (2.12)。

不可压条件在实际中并不能被严格满足。例如，气体可以被显著地压缩；液体的不可压性较好，但仍能被压缩，否则声音无法在水中传播。对计算机图形学中的模拟应用而言，不可压性几乎不会对视觉真实性造成影响，同时它极大的简化了 Navier-Stokes 方程组的形式，使得计算机科学家得以设计出相对简单、计算代价较小的模拟算法，因此几乎被所有模拟软件所采用。

## 2.2 平滑粒子动力学模型

平滑粒子动力学 (Smoothed Particle Hydrodynamics, SPH) 模型是流体模拟中一种典型拉式离散方法。Desbrun 和 Cani[9]最先把 Gingold 的平滑粒子流体动力学方法 (Smoothed Particle Hydrodynamics, SPH) 引入图形学中。随后，Müller 的工作[10]展现了 SPH 在实时动画和交互应用中的前景。SPH 将液体离散为固定数量的粒子。在最常见的情形中，每个粒子有一个统一的质量，支配空间中一块固定大小的球形区域。粒子携带着速度、密度、压力等流体参数，在 Navier-Stokes 方程组的支配下运动。

### 2.2.1 平滑核

平滑核规定了一个粒子对周围空间的影响范围和分布。平滑核 $W(\mathbf{r}, h)$ 是一个以原点径向对称的标量函数，其分布应当类似高斯分布[29]，但通常支撑集是有限的，并将支撑集的半径记为 $h$ 。本文中使用的核函数需要满足如下的条件，

- (1) 归一性条件，在支撑集上 $\int_V W(\mathbf{r}) d\mathbf{r} = 1$ 。
- (2) 高阶矩等于零， $\int_V W^n(\mathbf{r}) d\mathbf{r}$  ( $n \geq 2$ )。
- (3) 支撑集有限，对于某个 $h$ 有 $W(\mathbf{r}) = 0$  ( $\|\mathbf{r}\| > h$ )。这个条件使得计算空间某点的物理量时仅用考虑距离在 $h$ 范围内的粒子。本文针对这个特性设计了模拟加速数据结构，将在后文展开。
- (4) 非负性， $W(\mathbf{r}) > 0$ 。
- (5) 光滑可微分，SPH 计算中大量涉及平滑核上的微分。
- (6) 应能良好地避免聚集，下文将会对此展开描述。

光滑核的选取对 SPH 的稳定性、准确性和速度都会造成很大的影响。本文中使用了 Müller 在 2003[10]提出的 Poly6 核和 Spiky 核，它们具有不同的性质，分别用于 PBF 算法的密度估计和

订

线

梯度计算。Poly6 核的定义如下

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3, & 0 \leq r \leq h \\ 0, & otherwise \end{cases} \quad (2.14)$$

Poly6 核的梯度为

$$\nabla W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} -6(h^2 - r^2)^2 \hat{\mathbf{r}}, & 0 \leq r \leq h \\ 0, & otherwise \end{cases}$$

Poly6 核的梯度仅含有  $r$  的二次项，避免了计算向量长度时耗时的根号。同时应该注意到，Poly6 的梯度在靠近原点处逐渐消失。由于核函数的梯度常常用来计算粒子间的排斥力（例如，压强力  $\mathbf{F}_p = -\nabla p$ ），消失的梯度将会导致距离很近粒子之间无法排斥，发生聚集（Clustering）。在模拟过程中，当两个粒子一旦进入精度无法分辨的重合状态，将没有其他作用力能将它们分开，从而造成严重的体积损失现象。这显然背离了理想的模拟结果。

作为替代，本文使用 Spiky 核用于计算核函数的梯度。Spiky 核的定义如下

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3, & 0 \leq r \leq h \\ 0, & otherwise \end{cases}$$

Spiky 核的梯度为

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45}{\pi h^6} \begin{cases} (h - r)^2 \hat{\mathbf{r}}, & 0 \leq r \leq h \\ 0, & otherwise \end{cases} \quad (2.15)$$

注意到 Spiky 函数是一个向量函数，且在  $r = 0$  处函数值的长度达到最大。

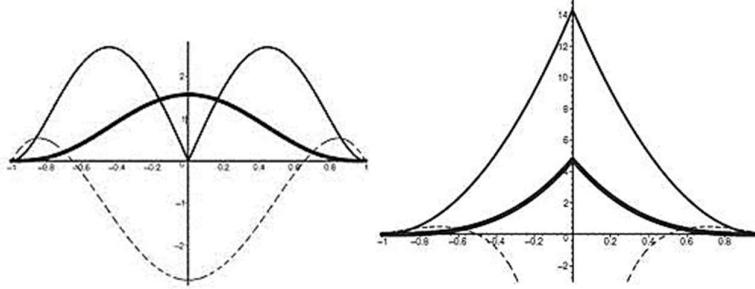


图 2.4 左图：Poly6 核函数，右图：Spiky 核函数；

粗线：函数值大小，细线：中心方向梯度大小，虚线：拉普拉斯算子大小[10]

### 2.2.2 SPH 流体参数

定义核函数后，空间中任意位置的一个物理量  $A(\mathbf{r})$  可通过对整个空间中粒子的核函数求和近似得到。

$$A(\mathbf{r}) = \sum_{i=1}^n \frac{A_i}{\rho_i} W(\mathbf{r} - \mathbf{r}_i) \quad (2.16)$$

定  
义  
？

通过定义 (2.16) 可以注意到，一个粒子中心的物理量  $A(\mathbf{r}_i)$  并不等于这个粒子携带的物理量  $A_i$ 。因为其他位置的粒子也会对这个位置有所贡献。

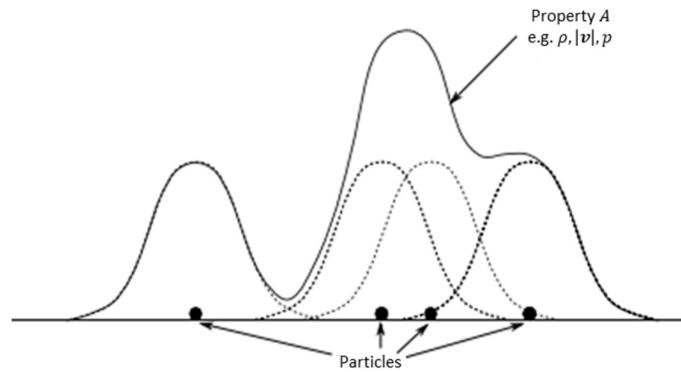


图 2.5 空间离散粒子还原空间连续物理量

因为模拟的物理量离散地记录在粒子上，本文通常不关心任意位置的物理量，而只关心粒子中心处的物理量。以下给出密度和压强的计算公式。若没有特别说明，速度等其他物理量都可以用类似方法算出。

粒子  $i$  位置的密度  $\rho(\mathbf{r}_i)$  为

$$\rho(\mathbf{r}_i) = \sum_{j=1}^N W(\mathbf{r}_{ij}) \quad (2.17)$$

其中  $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ 。SPH 模拟常常还需要计算物理量的梯度。式 (2.17) 为书写方便定义在  $\mathbf{r}_i$  上，但所有其他粒子的位置  $\mathbf{r}_j$  都是  $\mathbf{r}_i$  密度的隐含自变量。因此式(2.17)的梯度分为关于  $\mathbf{r}_i$  的梯度和关于  $\mathbf{r}_j$  的梯度。

$$\nabla_{\mathbf{r}_k} \rho(\mathbf{r}_i, h) = \begin{cases} \sum_{i=1}^N \nabla_{\mathbf{r}_i} W(\mathbf{r}_i - \mathbf{r}_j), & k = i \\ -\nabla_{\mathbf{r}_j} W(\mathbf{r}_i - \mathbf{r}_j), & k \neq i \end{cases} \quad (2.18)$$

根据热力学中的状态方程(Equation of State, EOS)，流体的密度和压强满足一些特定的关系。例如在完全气体假设，有  $p = \rho(\gamma - 1)e$ ，其中  $\gamma$  是一个跟气体比热容有关的绝热因子， $e$  是每单位质量的气体内能[30]。对于液体，计算机图形学通常使用可压缩性更弱的 Tait 方程。Tait 方程有如下的形式

$$p = \frac{k\rho_0}{\gamma} \left[ \left( \frac{\rho}{\rho_0} \right)^\gamma - 1 \right] \quad (2.19)$$

密度导出的压强是许多 SPH 方法中计算压强力  $\mathbf{F}_p$  的核心。但本文实现中使用的方法没有直

接应用到密度导出的压强，因此不再做展开。

### 2.2.3 SPH 流体模拟框架

SPH 拉式法的一个优势是，粒子物理量  $A$  的材料微分就是粒子的加速度，不需要处理材料微分的展开形式。实际上，考虑  $A(t, \mathbf{r})$  对时间的导数，其中  $\mathbf{r}$  是粒子位置，注意到  $\mathbf{r}$  也是时间  $t$  的函数，有

$$\begin{aligned}\frac{d}{dt}A(t, \mathbf{r}) &= \frac{\partial A}{\partial t} + \nabla A \cdot \frac{d\mathbf{r}}{dt} \\ &= \frac{\partial A}{\partial t} + \nabla A \cdot \mathbf{v} \\ &\equiv \frac{DA}{Dt}\end{aligned}$$

恰好是材料导数。

这个结论结合上一节中提到的状态方程导出压强的方法，本文给出 SPH 流体框架的一般形式，这个形式的 SPH 框架以力为改变流体状态的媒介，计算出压强力  $\mathbf{F}_p$ ，粘性力  $\mathbf{F}_v$  和重力  $\mathbf{F}_g$ ，直接利用动量方程更新速度  $\mathbf{v}$ ，最后根据不可压条件修正粒子速度和位置。

算法 2.1 SPH 流体框架的一般形式

---

```

for all particle  $i$  do
    find neighbors  $j$ 
for all particle  $i$  do
     $\rho_i = \sum_j m_j W_{ij}$ 
    compute  $p_i$  using  $\rho_i$ 
for all particle  $i$  do
     $\mathbf{F}_i^{pressure} = -\frac{m_i}{\rho_i} \nabla p_i$ 
     $\mathbf{F}_i^{viscosity} = m_i \mathbf{v} \nabla^2 \mathbf{v}_i$ 
     $\mathbf{F}_i^g = m_i \mathbf{g}$ 
     $\mathbf{F}_i(t) = \mathbf{F}_i^{pressure} + \mathbf{F}_i^{viscosity} + \mathbf{F}_i^g$ 
for all particle  $i$  do
     $\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t \mathbf{F}_i(t) / m_i$ 
     $\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t)$ 
for all particle  $i$  do
    apply incompressible constraint

```

---

装

订

$$d\mathbf{v} = \frac{\mathbf{F}}{m} \cdot dt$$

这个方法没有给出具体的不可压修正方法。实际上，它的原始并没有不可压性修正过程，能给出不真实但可信的模拟结果。这个框架广泛出现在包括[10]等早期具有广泛影响力的文献中，

线

是游戏等实时液体模拟应用中的标准方法。作为真实液体模拟的关键因素，本文并行化的特点之一，正是不可压性的求解过程。

#### 2.2.4 SPH 流体的特征和缺陷

SPH 相比流体模拟的欧式方法具有许多优势，包括

- (1) 没有模拟边界。欧式离散方法将空间分割为网格，由于计算能力限制，网格的空间尺度和分辨率之间具有不可调和的矛盾。而 SPH 中粒子的数量始终固定。  
(在本文和许多 SPH 的变种的实现中，为加速粒子邻居搜索引入了空间分割数据结构，使得本条失效)
- (2) 数值耗散小。欧式方法的数值耗散问题会使得没有粘度流体表现出有粘度流体的效果，能自动收敛到平衡状态。反之，SPH 通常需要额外引入粘性力使流体收敛。
- (3) 分辨率与密度近似正比。高密度区域因为流体聚集，需要更高的分辨率保证模拟精度。文献中有自适应分辨率的欧式方法，但由于实现复杂，计算代价大，未得到广泛应用。而 SPH 方法在高密度区域通常有更多地粒子，从而保证模拟效果。
- (4) 避免了欧式方法对没有流体的空间的不必要的计算。
- (5) 易于处理边界条件。

SPH 方法同时存在一些难以解决的问题，包括

- (1) **难以捕捉旋度较强的流体效果。**正因如此，SPH 通常用来模拟旋度效果较弱、体积联通成块的液体。
- (2) 需要修正不可压性。不可压方程是欧式方法求解的必要条件，因此能自动保证不可压性，而拉式方法的不可压性求解直到近几年才有突破性进展。
- (3) 密度有下限。因此不能很好捕捉低密度下的流体效果。

#### 2.3 求解不可压性的 PBF 方法

本文的模拟部分的主要工作是基于位置的流体（Position Based Fluids，PBF）在 GPU 上的并行实现。PBF 方法是 Macklin 等人在 2013 年 SIGGRAPH 图形学会议上发表的一种求解 SPH 流体不可压性的方法[15]。BPF 最显著的特点，是以直接以粒子位置分布为不可压性求解的目标。与之相比，预测-修正 SPH 方法（Predictive-corrective SPH，PCISPH）等传统的 SPH 方法则是形式化地引入压强力  $\mathbf{F}_p$  以预测粒子速度和位置。BPF 方式是 Müller 提出的基于位置的动力学（Position Based Dynamics，PBD）方法的一种推广。在 PBF 方法发表之前，PBD 已经被成功地运用在了包括布料、弹簧、刚体等一系列图形学物理模拟的领域中[31]。BPF 方法避免了

装

订

线

PCISPH 等方法中压强的累积，在较长的时间步长下表现出优越的鲁棒性。但其不可压性求解过程采用了雅可比方法实现并行，收敛速度较慢，因此需要更多的迭代次数以到达期望的不可压性效果，整体性能表现与 PCISPH 相似[16]。

本节将从理论角度出发，叙述 PBF 方法的主要过程。

### 2.3.1 流体位置的压强修正

PBF 方法是一种基于流体状态方程的方法（Equation of State, EOS）。在 2.2.2 节公式（2.19）中本文提到，密度可以用于导出压强，压强进而可以用于计算压强力，再利用动量公式积分更新粒子速度。PBF 方法观察到密度状态方程隐含了一个平衡条件，而这个平衡条件可以用于位置的更新。对粒子  $i$  定义  $C_i$  为 EOS 约束

$$C_i = \frac{\rho_i}{\rho_0} - 1 \quad (2.20)$$

其中  $\rho_0$  是液体的静止密度。当液体平衡时，必有  $\rho_i = \rho_0, p_i = 0$ 。这等价于对一块固定的液体，平衡时的体积保持不变，这是 2.1.2 节不可压条件（2.12）的直接推论。下文将展开 PBF 中粒子位置的迭代方法，通过对公式（2.20）求解，经过有限次数的迭代后必然能满足  $C_i = 0$ 。 Ci?

为了通过 EOS 方程（2.20）更新位置，首先注意到  $\rho_i$  是粒子位置的函数，因此  $C_i$  也是粒子位置的函数，设粒子个数为  $n$ ，有

$$C_i(\mathbf{p}_1, \dots, \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1 \quad (2.21)$$

$$\rho_i = \sum_{j \in \delta(\mathbf{p}_i)} m_j W(\mathbf{p}_i - \mathbf{p}_j) \quad \text{mj = ?} \quad (2.22)$$

其中  $\delta(\mathbf{p})$  是邻居算子，表示在以位置  $\mathbf{p}$  为中心，核半径为大小的区域内的粒子的集合。为计算方便本文假设粒子质量都为 1 千克，即  $m_i = 1, \forall i$ 。本文期望经过一个粒子位置修正  $\Delta\mathbf{p} = (\Delta\mathbf{p}_1, \dots, \Delta\mathbf{p}_n)$  后，每个粒子的 EOS 约束能被满足，即

$$C(\mathbf{p} + \Delta\mathbf{p}) = \begin{pmatrix} C_1(\mathbf{p}_1 + \Delta\mathbf{p}_1, \dots, \mathbf{p}_n + \Delta\mathbf{p}_n) \\ \vdots \\ C_n(\mathbf{p}_1 + \Delta\mathbf{p}_1, \dots, \mathbf{p}_n + \Delta\mathbf{p}_n) \end{pmatrix} = \mathbf{0} \quad (2.23)$$

可应用梯度下降法，让  $\Delta\mathbf{p}$  与  $C$  在  $\mathbf{p}$  处的梯度  $\nabla_{\mathbf{p}} C(\mathbf{p}) = (\nabla_{\mathbf{p}} C_1, \dots, \nabla_{\mathbf{p}} C_n)^T$  平行，设学习率  $\lambda = (\lambda_1, \dots, \lambda_n)^T$ 。一种朴素的方法是，设定  $\lambda$  为某个预设值，迭代更新粒子位置，直到方程（2.23）达到某个精度阈值。这种方法在实际中并不可行，因为  $\lambda$  的预设值无法准确估计。静态的学习率也可能导致收敛速度缓慢。步长

实际上可以利用方程（2.23）的一次微分展开估计  $\lambda$ 。注意到方程（2.23）可微，有一次近似

$$C(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla C^T \Delta\mathbf{p} = 0 \quad (2.24)$$

$$\approx C(\mathbf{p}) + \nabla C^T \nabla C \lambda = 0$$

以 $\lambda$ 为未知数的方程含有 n 个标量方程和 n 个标量未知数，是一个适定性问题。由线性代数知识可知，若 $C(\mathbf{p})$ 可逆，则方程有确定解。但是由于 $C$ 的维度是 n，在实际中通常大于 $10^4$ ，而最好的的矩阵求逆算法复杂度高达 $O(n^{2.373})$ ，完全无法承受，因此也不能通过直接求解估计 $\lambda$ 。

作为方程 (2.24) 的近似解法，本文考虑高阶线性方程组的雅可比雅可比迭代法[32]。雅可比迭代法每步使用上次迭代的近似解求解下一个近似解，且每次只求解一个未知量，n 个未知量的求解互不干扰。使用雅可比迭代法的更新公式为

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_{j \in \delta(\mathbf{p}_i)} \|\nabla_{\mathbf{p}_j} C_i\|} \quad (2.25)$$

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta \mathbf{p} = \mathbf{p} + \nabla C \lambda \quad (2.26)$$

同时这里给出 $\nabla_{\mathbf{p}_j} C_i$ 的公式。

$$\nabla_{\mathbf{p}_j} C_i = \frac{1}{\rho_0} \sum_{k \in \delta(\mathbf{p}_i)} \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_k) \quad (2.27)$$

根据 i 和 j 的相同和不同， $\nabla_{\mathbf{p}_j} C_i$ 又有两种形式

$$\nabla_{\mathbf{p}_j} C_i = \frac{1}{\rho_0} \begin{cases} \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_k), & j = i \\ -\nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j), & j \neq i \end{cases} \quad (2.28)$$

公式(2.25)的分母位置可能为零，这会导致运算出现 NaN 错误。一种情况是没有邻居的孤立粒子，这种情况下分子其实也为零，可以通过判断 $\delta(\mathbf{p}_i)$ 是否为 0 避免 NaN。还有一种情况是当粒子的邻居位于核半径的边缘，这时 Spiky 核的给出的梯度将会很小，而由于梯度处在分子位置，将会导致结果很大，产生严重的误差。表现在模拟结果上，则是粒子在远离邻居的过程中会突然受到一股“弹射”的力。这与物理世界是严重相悖的。因此有必要对公式 (2.25) 做修正。本文统称这个问题为孤立粒子问题。

PCISPH 也存在孤立粒子的问题。PCISPH 算法在出现孤立粒子时，假想地在粒子周围填充一定地邻居，保证梯度和不为零。在此本文参照[15]中的办法，直接在分母处引入一个（相对梯度和）小的常数 $\epsilon$ 。在本文的最终实现中取 $\epsilon = 1000$ 。这个值在数值上并不小，但相对于 Spiky 梯度和来说是小的。修正后的 $\lambda$ 计算公式为

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_{j \in \delta(\mathbf{p}_i)} \|\nabla_{\mathbf{p}_j} C_i\| + \epsilon} \quad (2.29)$$

公式 (2.26) 对粒子 i 仅根据 $\lambda_i$ 和 $C_i$ 更新其位置。换句话说，仅根据粒子 i 本身的信息，而没有将它的邻居的更新量一并考虑。为了使得更新公式满足牛顿第三定律，对公式(2.26)略作修改，

订

不仅考虑粒子*i*本身，也考虑邻居粒子的更新量。这个技巧在 SPH 模拟中被广泛的使用。改进的粒子位置更新公式为

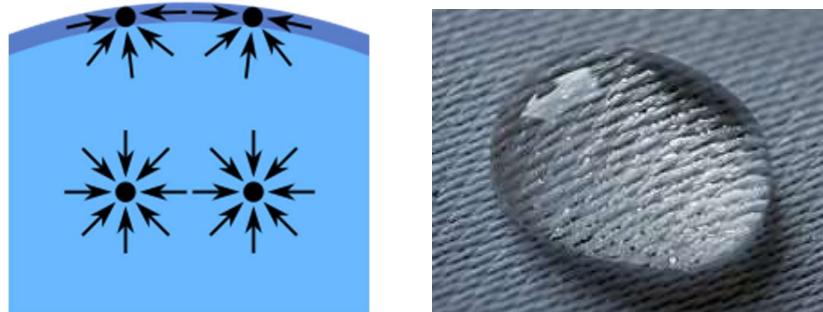
$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \frac{1}{\rho_0} \sum_{j \in \delta(\mathbf{p}_i)} (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j) \quad (2.30)$$

在实际的求解中，因为计算能力的限制，不可能迭代过多，仅要求能满足视觉真实性即可。本文的实现中每帧设定迭代 4 次，更多的迭代次数对改善模拟结果帮助不大。

除了上文中提到的参数，本文在实现中设定的其他常数参数包括核半径  $h = 0.1m$ ，静止密度  $\rho_0 = 8000kg/m^3$ ，重力  $g = 9.8m/s^2$ ，模拟时间步长  $dt = 0.0083$ （近似 120Hz）。

### 2.3.2 流体速度的表面张力修正

2.1 节中给出的 Navier-Stokes 方程描述了连续流体的物理规律，但没有给出流体的边界条件。液体的表面张力性质，就是一种典型的边界条件。在液体-气体的表面处的液体分子，受到单侧其他液体分子的分子间作用力的作用（范德华力或氢键等），产生了一个法向的表面张力。而内部分子由于四周被其他分子包围，则没有表面张力。小水珠在表面张力的作用下，表面呈弧形。



装

图 2.6 表面张力形成原理

仿照表面张力的形成原理，可以在 SPH 中通过引入分子间作用力实现表面张力的效果，论文[33]等称这个力为“虚拟压强”（Artificial pressure）。与真正的压强力不同，它不是通过压强计算得到的，而是通过直接估计粒子和邻居的距离得到的，不具有物理真实性，但是经验中能够得到良好的模拟效果。

在很多 SPH 方法中，虚拟压强可以防止粒子出现聚合现象[34][35][36]。边界粒子密度被低估是粒子聚合线性的主要原因。孤立的一个或者一簇粒子由于周围没有邻居粒子，通常处于密度达不到静止密度的状态，因此动量方程会使得它们聚集成很小的一簇。在 PBF 中这个问题更加明显。没有引入虚拟压强的情况下，两个孤立粒子在某个时刻靠近会进入几乎重叠的状态，它们的相对位置保持不变。若它们有丝毫的位置偏离，立即会有很高的吸引力使它们回归近似

订

线

重叠的静止状态。根据静止密度的不同，三个及以上粒子也可能会进入这种状态，造成**体积坍缩**（Loss of volume）。

为了应对体积坍缩的问题，[15]中引入了最早在[33]中提出的虚拟压强方法。这种方法根据两个粒子的距离和一个理想最小距离的比例对距离太近的粒子施加惩罚性的压强力。对粒子*i*和粒子*j*，令虚拟压强系数 $s_{\text{corr}}$ 为

$$s_{\text{corr}} = -k_{\text{corr}} \left( \frac{W(\mathbf{p}_i - \mathbf{p}_j)}{W(\Delta q)} \right)^{n_{\text{corr}}} \quad (2.31)$$

并且将这个力直接作用在粒子的位置更新上。注意到，此处的力不是牛顿力学意义上的力，而是促使流体状态改变的一种因子。加入虚拟压强修正的公式（2.30）变为

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \frac{1}{\rho_0} \sum_{j \in \delta(\mathbf{p}_i)} (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla W(\mathbf{p}_i - \mathbf{p}_j) \quad (2.32)$$

在实现中本文取 $\Delta q = 0.03h$ ,  $n = 4$ ,  $k_{\text{corr}} = 0.001$ ,  $n_{\text{corr}} = 4$ 。

### 2.3.3 流体的粘度修正

为了还原动量方程中的粘性力 $\mathbf{F}_v$ ，本文引入对粒子速度引入一个由相对速度引起的粘度修正。粘度修正正是 Navier-Stokes 方程中粘性力的近似。

$$\mathbf{v}_i^{\text{new}} = \mathbf{v}_i + c_{\text{XSPH}} \sum_{j \in \delta(\mathbf{p}_i)} \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_i + \rho_j} W(\mathbf{p}_i - \mathbf{p}_j) \quad (2.33)$$

公式（2.33）还原了牛顿流体粘性力公式（9）的两个主要性质，一是粘性力大小于正比于粒子相对速度，二是在一微小面元上粘性力大小正比于积分面积。第一条性质来源于分子处的粒子速度差；第二条性质来源于分母处的密度和，在粒子质量固定的假设下，流体微元表面积随密度增大而减小。严格的来讲，微元表面积 $dS$ 与密度和 $\rho$ 满足 $dS \propto \rho^{-2/3}$ （因为面积正比于二次直径，体积正比于三次直径）。但此处简单的使用密度的倒数，可以避免代价很大的指数运算，同时也能提供较为真实的模拟效果。

SPH 的粘度修正对于维持液体的运动一致性具有非常重要的作用。它使得相邻的粒子倾向于成簇运动，同时耗散流体动能，使得流体最终进入平衡状态。没有或过小的粘度修正时，流体粒子可能会在数值误差的作用下产生抖动。

公式（2.33）在分母位置引入的密度和是对[15]粘度修正公式的一个改进。这个改进是在多次实验中得到启发的。在视觉上，未考虑密度的粘度修正会导致液体表现出区域性的粘度分歧。

在实现中本文取粘度系数 $c_{\text{XSPH}} = 1$ 。

装

订

线

## 2.4 PBF 方法的 GPU 并行化实现

本节叙述 PBF 方法的 GPU 并行化算法的流程、设计与实现。本文首先概述 PBF 方法的算法流程，指出 PBF 高度适合并行化计算的特征，并对邻居查找、迭代方式、边界处理等几个关键设计做简要说明。2.4.2 节介绍通用 GPU 计算（General Purpose GPU, GPGPU）和本文使用的 Nvidia CUDA 并行语言。2.4.3 节介绍 CUDA 核函数和数据结构设计。2.4.4 节介绍本文使用的哈希网格邻居查找算法，2.4.5 节对算法管线的实现细节做进一步说明。

### 2.4.1 实现管线概述

本文的 PBF 算法的伪代码如算法 2.2 中所示。右侧是实现中伪代码对应的 CUDA 核函数。核函数是通用 GPU 计算的计算单元，2.4.2 节将会对这个概念进行说明。

生成初始粒子后，PBF 方法以一帧为单位，从上一帧状态出发，模拟得出下一帧状态，并以此反复循环。在一帧的模拟中，粒子首先使用蛙跳积分法（Leapfrog integration）交替更新速度和位置。这个步骤被形象的称为流体对流（Advection）。根据更新后的粒子位置建立用于邻居查找的哈希网格数据结构[37]。之后，根据公式（2.32）迭代地使用雅可比方法对粒子位置进行修正。最后，对粒子的速度进行粘度修正。

算法 2.2 PBF 模拟算法流程

Algorithm 2 Simulation Flow	Kernels
1: Initialize $\mathbf{p}_i$ for all particle $i$	
2: <b>for all</b> particle $i$ <b>do</b>	
3: $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_g$	advectKernel
4: $\mathbf{p}_i \leftarrow \mathbf{p}_i + \Delta t \mathbf{v}_i$	getGridId
5: <b>end for</b>	sortByGridId
6: Build hash grid $\delta(\mathbf{p}_i)$ for all particle $i$	computeGridRange
7: <b>loop</b> solver_iterations <b>do</b>	
8: <b>for all</b> particle $i$ <b>do</b>	
9:     calculate $\lambda_i$	computeLambda
10: <b>end for</b>	
11: <b>for all</b> particle $i$ <b>do</b>	
12:     calculate $\Delta\mathbf{p}_i$	computePos
13:     update $\mathbf{p}_i^{new} \leftarrow \mathbf{p}_i + \Delta\mathbf{p}_i$	
14:     handle collision with boundary	
15: <b>end for</b>	
16: <b>for all</b> particle $i$ <b>do</b>	
17:     update $\mathbf{v}_i \leftarrow \mathbf{p}_i^{new} - \mathbf{p}_i$	updateVelocity
18:     apply viscosity confinement	computeXSPH
19: <b>end for</b>	
20: $\mathbf{p}_i \leftarrow \mathbf{p}_i^{new}$ for all particle $i$	
21: <b>end loop</b>	

装

订

线

下面对算法中使用的几个关键设计做进一步说明。

**蛙跳积分法：**本文首先使用欧拉积分，更新重力引起的粒子速度变化。再使用欧拉积分更新新速度下的粒子位置。这种两步积分方法称为蛙跳积分法（Leapfrog integration）[38]。蛙跳积分法是一个二阶积分方法，它的引入总误差正比于时间步长的平方，能给出相比欧拉方法等一阶方法更小的误差。

**邻居查找：**SPH 方法的粒子核半径  $h$  是有限的，两个中心距离大于  $h$  的粒子对互相没有作用，因此产生了粒子邻居的概念。记  $\delta(p)$  是中心位置在  $p$  的粒子的邻居几何，任意邻居到  $p$  的距离不超过  $h$ 。在利用公式 (2.16) 计算粒子的物理量时，仅需要考虑粒子的邻居即可。SPH 方法的粒子数量通常在  $10^4$  量级，一个核直径内的粒子数量通常小于  $10^2$  量级，因此预先计算好粒子的邻居，能极大的减小位置核速度修正的计算量。

计算粒子邻居的代价较为高昂，本文的算法每一帧仅计算一次邻居。注意到，位置修正有多次迭代。第一次迭代后，预计算好的邻居就不能精确反应真实的邻居状态了。更具体的考虑一个粒子的邻居，有的邻居将退出邻居集合，其他的粒子可能加入邻居集合。总体来看预计算邻居集合在第一次迭代后是真实邻居集合的子集，或者说是真实邻居集合的一个低估（Underestimate）。

在 PCISPH 方法中，低估的邻居集合会导致错误的压强，甚至会产生负压强。如果邻居集合不能得到及时更新，错误的压强将会累计，导致不真实的模拟结果。因此 PCISPH 每一轮迭代都需要重新计算邻居。PBF 则没有压强累计的过程。相反，低估的邻居只会减弱位置修正的幅度。反应在结果中，PBF 方法表现出一种类似粘性力的流体阻尼（damping）效果，因此具有较好的鲁棒性。

**雅可比方法：**在 2.3.1 节中本文已经介绍过雅可比方法，并给出了使用雅可比方法修正粒子位置的公式。雅可比方法是经典的线性方程组定常迭代求解法，属于同类型的解法还有 Gauss-Seidel 法和超松弛（Successive over-relaxation, SOR）法等[32]。雅可比方法相比其他迭代法的优势在于它非常适合并行化。雅可比方法中，每个变量每轮迭代仅依赖上一轮迭代中所有变量的结果。与之相对的，Gauss-Seidel 迭代法中，每个变量在一轮迭代中有前后依赖的关系，因此只能串行化处理。考虑到 SPH 粒子的数量高达  $10^4$ ，若能讲针对每个粒子  $\lambda_i$ ,  $\mathbf{p}_i$  的计算和速度修正做并行化处理，将会给性能带来巨大的提升。现代 Nvidia GPU 有上千个 CUDA 计算核单元，能够提供很好的并行加速效果。因此本文选择了使用 Nvidia CUDA 实现 PBF 的并行版本。

**边界处理：**SPH 中的边界处理主要指粒子与刚体边界的碰撞。除此之外，有的文献针对 2.3.2 节中提到的边界粒子密度被低估问题，提出了边界虚拟粒子等方法[39]补偿边界密度。最初本文的实现在刚体边界上有粒子抖动的问题，因此引入了[39]中的方法。在本文加入了粘性力后，粒子抖动现象消失。经试验，边界密度补偿对模拟结果改善不大，因此本文最终仅引入了刚体边界碰撞这一种边界条件。

装

订

线

本文的刚体边界是与坐标轴对齐的一个立方体，流体被限制在立方体内运动。粒子位置在不可压修正迭代的过程中，如果超出了立方体边界，则会被投影到最近的立方体表面的内侧，且保持与边界保持一个很小的距离 $\epsilon_B$ 。实现中本文取 $\epsilon_B = 10^{-3}\text{m}$ 。这个形式的边界处理可以看作是粒子与刚体边界发生了完全非弹性碰撞。这个处理似乎不符合真实世界中水与刚体表面碰撞产生的飞溅的效果。然而，考虑一簇粒子与刚体发生的碰撞：首先发生碰撞的粒子将贴紧刚体表面，并不再沿法向方向移动；后来的粒子与先发生碰撞的粒子距离缩小，产生很高的密度，进而会在不可压修正的迭代中被反向排斥。这十分符合物理世界中的现象——少量水滴碰撞后粘滞在刚体表面，水量丰富后才会发生反弹。

注意到算法在流体对流和邻居查找步骤中是不做边界限制的。粒子对流后可能超出边界。下一节本文会提到，用于哈希网格对立方体空间做均等的分割，要求粒子处在边界内。此时本文把越界的粒子归在投影后的网格内，但不做位置上的更新。

本文的边界形式比较简单。若要处理流体与任意形状的刚体碰撞，有两种流行的方法。一种是将刚体离散成带符号的距离场（Signed Distance Field）[5]，使用体素记录具体场函数。位置修正时，利用查询距离场得到的粒子所在处的距离值，将处于刚体内部的粒子投影到粒子边界上。另一种 PBF 方法特有的做法，则是将刚体离散成粒子，在 PBD 的框架内处理粒子间约束[40]。这种方法更容易实现流体和刚体之间的双路交互（Two-way coupling），实现刚体在流体冲击下移动的效果。

#### 2.4.2 CUDA 并行计算概述

Nvidia 在 2006 年 11 月推出了 CUDA 通用并行计算平台。CUDA 最初是统一计算设备架构（Compute Unified Device Architecture）的缩写，它提供了一个类 C/C++ 的编程环境和一套并行计算 API。使用 CUDA 编写的程序将被编译成 Nvidia GPU 指令集，在 GPU 上进行运算。

在 GPU 与 CPU 有截然不同的架构。以 Fermi 架构编号为 G80 的 GPU 为例（GeForce 8800），这款 GPU 拥有 16 个流式多处理器（Streaming multiprocessor, SM），每个 SM 又有 32 个 CUDA 核心，每个核心拥有一个单精度（32 位）浮点运算单元和一个整数运算单元。每个 SM 拥有两个 warp 调度和分配器，共享一些寄存器，一个 L2 缓存和一个 64KB 可配置的 L1 缓存。一个 warp 是 32 个执行相同 GPU 指令的线程的集合[41]。因此，Fermi GPU 可同时执行 512 个线程。相比之下，CPU 能够同时执行的线程取决于核心数。民用 CPU 通常不会超过 8 个核心。在高线程并发下，CPU 线程上下文切换带来的代价将会抵消掉并行化带来的速度提升。

CUDA 提供了统一的编程模型启动和调度 GPU 任务。GPU 任务以用户编写的核函数（Kernel）为执行单位，由 CPU 发起，在 GPU 上异步地执行。CUDA 将核函数的并发组织成 Grid, Block 和 Thread 三个层次，每个核函数开启的所有线程称为一个 Grid。一个 Grid 内有若干 Block，每个 Block 内又有若干 Thread。Block 和 Thread 各自都可以是一维、二维或是三维结构。

装

订

线

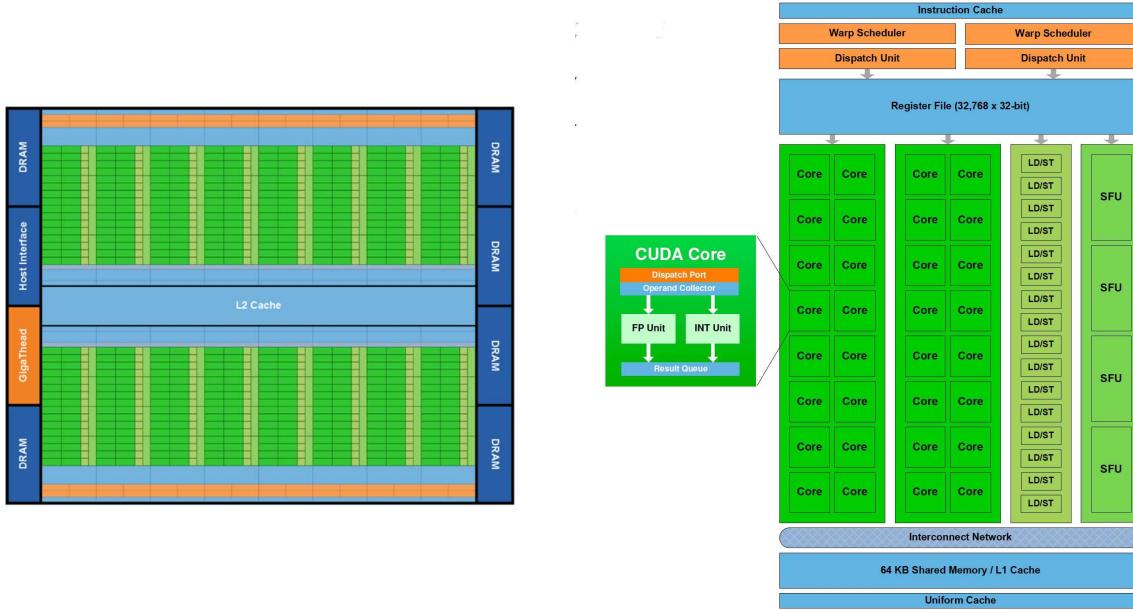


图 2.7 Fermi GPU 架构[42]

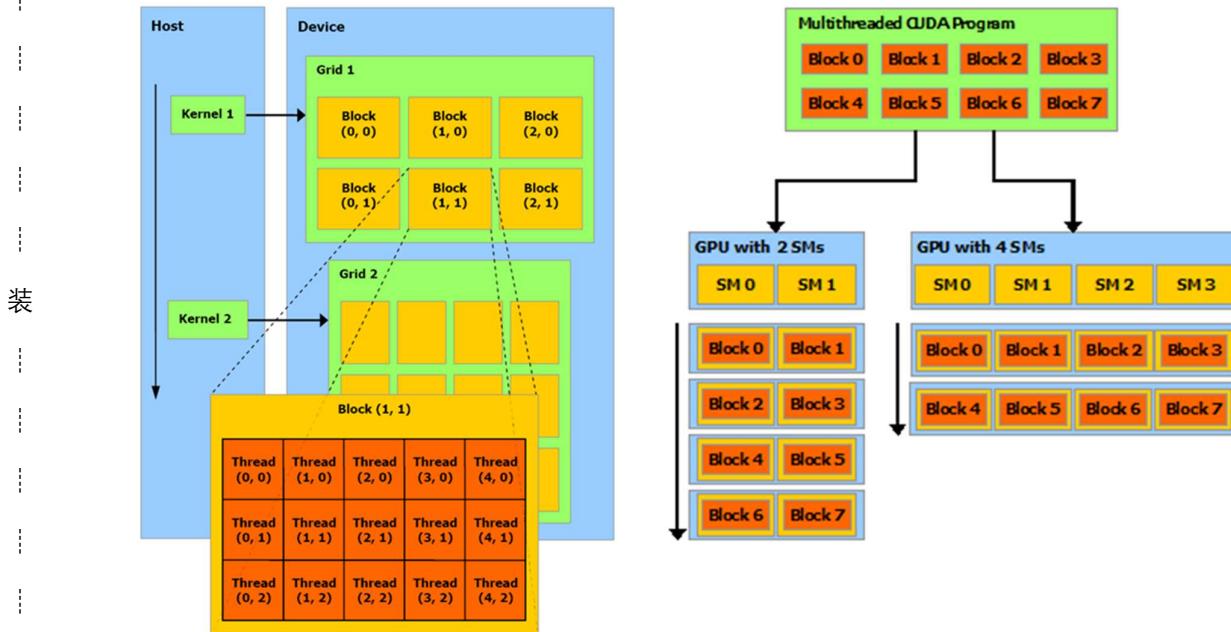


图 2.8 CUDA 核函数组织和执行层次[43]

左：并行组织；右：执行分配

在 GPU 层面，核函数线程以 Block 为单位进行计算资源的分配。一个 SM 同一时刻只能执

行一个 Block，每个 Block 内的线程以 32 个为一个 Warp 统一调度。一个 Warp 调度器只有一个程序计数器（Instruct Pointer），因此每个 Warp 内的线程同时执行核函数的同一语句。因为这个限制，若一个 Warp 之内的线程在分支语句出现了分歧，则必须一部分线程停下来等待另一部分线程执行。

#### 2.4.3 CUDA 核函数和数据结构设计

以下列出 PBF 模拟使用的所有数据结构。在粒子位置初始化时，由 CPU 把数据从内存拷贝到显存上。之后所有数据都以数组的形式存在 GPU 显存上，并在 CPU 管理的 Simulator 类中保留一个指针。这些数组有的通过 CUDA 分配，有的则通过 OpenGL 分配，并在模拟算法时运行动态地在 CUDA 中绑定。后者是为了方便地渲染模拟结果，因为 CUDA 提供了绑定 OpenGL buffer 和 texture 的 API，但反过来没有提供使用 OpenGL 读取 CUDA 数据的方法。同样因为这个限制，本文在渲染部分还原液体表面时没有选择 CUDA，而是选择了使用面元着色器（Fragment Shader）作为更简便的计算手段。

表 2.1 PBF 算法用数据结构

名称	类型	位置	描述
<code>dc_pos</code>	<code>float3 *</code>	OpenGL	粒子位置
<code>dc_npos</code>	<code>float3 *</code>	OpenGL	备用粒子位置
<code>dc_tpos</code>	<code>float3 *</code>	CUDA	临时粒子位置
<code>dc_vel</code>	<code>float3 *</code>	CUDA	粒子速度
<code>dc_nvel</code>	<code>float3 *</code>	CUDA	备用粒子速度
<code>dc_iid</code>	<code>uint *</code>	OpenGL	粒子编号
<code>dc_gridId</code>	<code>uint *</code>	CUDA	粒子所属网格编号
<code>dc_gridStart</code>	<code>uint *</code>	CUDA	网格起始索引*
<code>dc_gridEnd</code>	<code>uint *</code>	CUDA	网格终止索引*
<code>dc_lambda</code>	<code>float *</code>	CUDA	粒子所在处 $\lambda$ 值
<code>dc_rho</code>	<code>float *</code>	CUDA	粒子所在处 $\rho$ 值

\* 哈希网格相关数据结构。将在 2.4.4 节中做进一步说明。

以下按照算法的前后逻辑顺序列出 PBF 模拟使用的所有核函数。这其中有的核函数使用了 thrust 并行库。thrust 封装了一些常见的并行任务，例如数据的一对一映射（`thrust::transform`），拷贝（`thrust::copy_n`），排序（`thrust::sort_by_key`）和求最值（`thrust::minmax_element`）等。

2.4.1 节中算法 2 列出了伪代码和 CUDA 核函数的对应关系。

表 2.2 PBF 算法用 CUDA 核函数

模块	名称	输入	输出	描述
<code>advect</code>	<code>advect_kernel</code>	<code>dc_pos, dc_vel</code>	<code>dc_npos</code>	Leapfrog 积分
<code>buildGridHash</code>	<code>thrust:::</code>	<code>d_npos</code>	<code>d_gridId</code>	计算粒子所属网格编号

transform			
thrust:: sort_by_key	d_gridId, d_pos, d_vel, d_npos, d_nvel, d_iid	d_gridId, d_pos, d_vel, d_npos, d_nvel, d_iid	根据网格编号排序粒子
	computeGridRange	dc_gridId	dc_gridStart, dc_gridEnd
<b>correctDensity</b>	computeLambda	dc_gridId, dc_gridStart, dc_gridEnd, dc_npos	计算粒子 $\lambda, \rho$
	computetpos	dc_lambda, dc_gridId, dc_gridStart, dc_gridEnd, dc_npos	计算粒子 $\Delta p$
	thrust:: copy_n	d_tpos	更新 $p \leftarrow p + \Delta p$
<b>updateVelocity</b>	thrust:: transform	d_pos, d_npos	计算 $v$
<b>correctVelocity</b>	computeXSPH	dc_rho, dc_gridStart, dc_gridEnd, m_gridHashDim, dc_npos, dc_vel,	修正 $v$

#### 2.4.4 哈希网格邻居寻找算法

在 2.4.1 节本文提到邻居寻找算法是 SPH 的一个重要组成部分。邻居查找算法一般由预处理算法和 动态查询算法两个步骤组成。本文使用的是哈希网格（Hash grid）邻居查找算法[37]。哈希网格算法将粒子活动的立方体空间等大地分割成许多直径为核半径 $h$ 的立方体，称为格子（Grid）。空间中的每个粒子唯一的属于一个格子。一个格子中的所有粒子，通过一个哈希函数，从粒子位置映射到格子格子编号上。

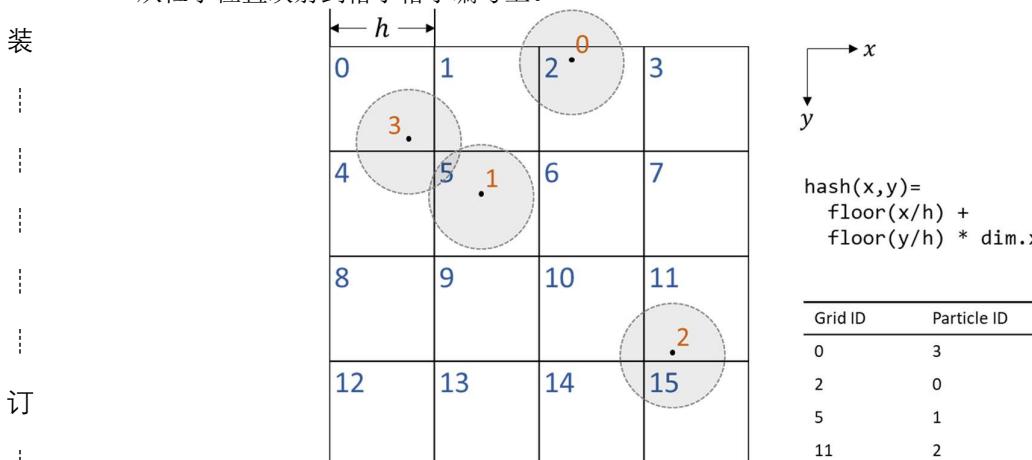


图 2.9 哈希网格结构

上图展示了一个二维哈希网格。这个网格编号从左到右，从上到下依次排布。`dim.x` 是 x 方向网格的维度。这个结构很容易推广到三维地形式，也是本文的实现中使用的形式。邻居寻找算法通常需要查询一个格子周围的数个格子。因此其他形式的哈希函数，例如将格子编号按照 Z 字形排布的哈希函数，可能对 GPU 缓存更加友好。但在此本文不做探究。

在哈希网格算法的预处理阶段，算法首先对每个粒子计算它所属的格子编号，再对每个格子计算它所含的粒子集合。计算格子所含粒子集合有两种做法，一种做法对每个格子记录 `count` 和 `cells` 两个变量，分别表示粒子个数和粒子编号数组。在计算粒子的格子编号时，使用原子操作更新所属格子的 `count` 和 `cells`。

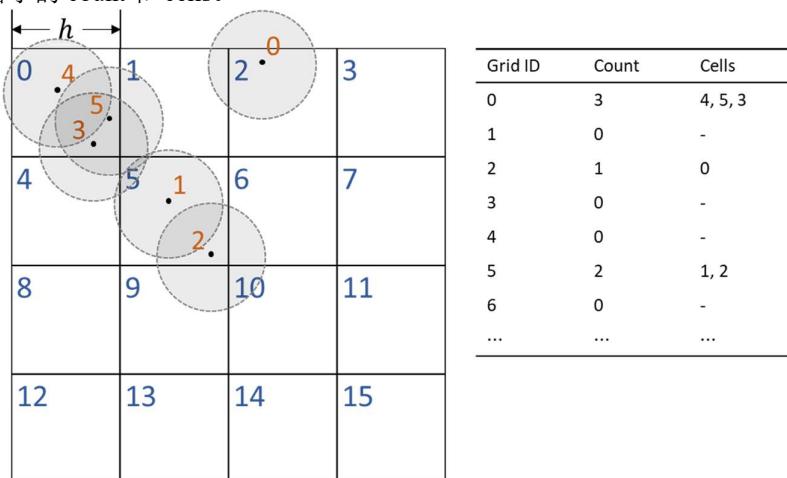


图 2.10 原子更新格子记录

另一种做法首先将粒子根据所属格子所属编号进行排序。然后计算每个格子所含粒子的起始编号 `gridStart` 和结束编号 `gridEnd`。这里使用了左闭右开区间，因此编号为 `gridEnd[id]` 的粒子不属于编号为 `id` 的格子。排序使用了 thrust 库的 GPU 基数排序算法。

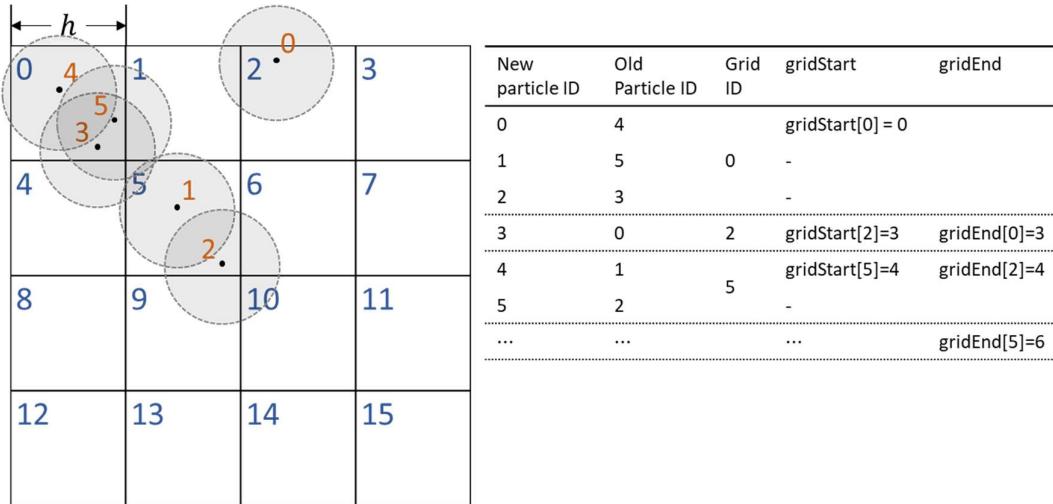


图 2.11 排序后计算格子记录

为了计算 gridStart 和 gridEnd, 对排序后的每个粒子比较它和前一个粒子的所属格子编号。若格子编号不同, 说明当前粒子是当前格子的起始编号, 也是前一个粒子所属格子的终止编号。由于每个格子的 gridStart 和 gridEnd 只会被更新一次, 这个方法不需要原子操作。虽然比上一个方法多了一步排序操作, 性能却表现得更好。负责这个任务的是核函数 computeGridRange。

当需要查询一个粒子的邻居时, 首先根据粒子位置计算粒子所在格子编号。粒子所有可能的邻居都将位于所在格子以及相邻的格子里。因此在三维空间中, 共有上下、左右、前后和中心共 7 个格子需要考虑。使用 gridStart 和 gridEnd 中记录的格子所含粒子编号, 即可遍历得到当前粒子所有可能的邻居。

#### 2.4.5 实现细节

装

**可调用对象传参:** 2.4.3 节中的表 X 只列出了传入核函数的数组。实际传入核函数的还包括常数参数  $h, k_{corr}$  等。computeLambda, computetpos 和 computeXSPH 这三个核函数需要计算 Poly6 核和 Spiky 核的梯度, 这是通过向核函数传入一个可调用对象 (Callable Object) 实现的。例如, 定义 Poly6 的可调用结构体为如下的形式:

代码 2.1 Poly6 可调用结构体

```
#define M_PI 3.14159265359
struct getPoly6 {
    float coef, h2;
    getPoly6(float h) {
        h2 = h * h;
        float ih = 1.f / h;
        float ih3 = ih * ih * ih;
        float ih9 = ih3 * ih3 * ih3;
        coef = 315.f * ih9 / (64.f * M_PI);
```

线

---

```

    }
    __device__
    float operator()(float r2) {
        if (r2 >= h2) return 0;
        float d = h2 - r2;
        return coef * d * d * d;
    }
};
```

---

可调用对象的类型是比较复杂的。为了书写简便，可将传入核函数的可调用参数设为模板类型。调用核函数时，传入动态的生成一个可调用对象，编译器能够自动特化得到正确的变量类型。

代码 2.2 含可调用参数的核函数的定义和调用

---

```

/* Callable parameter definition */
template <typename Poly6F>
__global__
void computeXSPH(..., Poly6F poly6, ...) { ... }

/* Pass Callable parameter */
computeXSPH(..., getPoly6(h), ...);
```

---

另一种做法是直接在核函数内定义 Poly6 和 Spiky 梯度的计算函数（CUDA \_\_device\_\_ 函数）。这也是本文最初的做法。观察 Poly6 的定义可以发现，当  $h$  确定时， $W_{poly6}(\mathbf{p}) = C \cdot g(\mathbf{p})$ ，其中  $C$  是某个只与  $h$  有关的常数。利用可调用结构体，可以在传入核函数前确定好常数，从而节省核函数大量的计算时间。本文通过实验发现这个优化能带来超过 50% 的帧数提升。

**双缓冲：**本文对位置和速度使用了双缓冲方法，因此它们有两套变量。其中一套用于记录一帧模拟开始时粒子状态，另一套则是模拟算法写入的对象。在每一帧模拟开始时，调用 Simulator 类的 void step(uint d\_pos, uint d\_npos, uint d\_vel, uint d\_nvel, uint d\_iid, int nparticle) 函数注册 OpenGL 管理的缓存，将其映射为 CUDA 核函数可访问的指针。模拟算法以 d\_pos 和 d\_vel 为上一次模拟的位置和速度结果，将本次模拟结果写入到 d\_npos 和 d\_nvel 中。使用双缓冲可以避免算法 2 最后一步从  $\mathbf{p}_{new}$  到  $\mathbf{p}$  的拷贝，节省了一定的时间。渲染模拟结果时，本文也需要从两个位置缓冲中选择较新的一个作为渲染的对象。算法 2 的 18 行对粒子速度进行涡度和粘度修正隐含了一个对粒子速度的并发读写冲突，因此需要一个中间缓冲区 tpos。修正后的速度首先被写入 tpos 中。当所有粒子修正结束后，再把 tpos 写回 npos。

**核函数指令优化：**在 2.4.2 节中本文提到，一个 warp 中 32 个线程仅有一个程序计数器，因此只能同时执行一条语句。当线程中出现分支分歧时，一部分线程将会挂起，等待另一部分线程的完成。因此有必要尽量减少核函数中的分支语句，以提高 GPU 的利用率。

为了尽量减少分支，GPU 对一些常见的指令进行了去分支的设计。例如 Nvidia GPU 上的最大最小值是通过 IMNMX 和 VMNMX 指令完成的。x86 CPU 的 SSE 指令集亦有类似功能的指令，

但编译器并不会默认生成。

核函数寻找粒子邻居需要遍历粒子所在格子的 6 个邻居格子。邻居格子可能会超出立方体边界。这些越界邻居格子是没有粒子的，gridStart 和 gridEnd 中也没有它们的记录，因此需要跳过。一种方法是，在每个维度上判断越界，尽早跳过。另一种方法则是在最内侧循环判断越界，尽量迟跳过。这两种方法在效果上是等价的，在性能上有差异。后者因为尽量减少了核函数分支分歧，能够获得更好的性能。

代码 2.3 尽量早和尽量迟越界判断

```
/* Late out-of-bound condition */
for (int dx = -1; dx <= 1; dx++)
for (int dy = -1; dy <= 1; dy++)
for (int dz = -1; dz <= 1; dz++) {
    int x = ind.x + dx, y = ind.y + dy, z = ind.z + dz;
    if (x < 0 || x >= cellDim.x ||
        y < 0 || y >= cellDim.y ||
        z < 0 || z >= cellDim.z)
        continue;
    /* ... */
}
/* Early out-of-bound condition */
for (int dx = -1; dx <= 1; dx++) if (x >= 0 && x < cellDim.x)
for (int dy = -1; dy <= 1; dy++) if (y >= 0 && y < cellDim.y)
for (int dz = -1; dz <= 1; dz++) if (z >= 0 && z < cellDim.z) {
    int x = ind.x + dx, y = ind.y + dy, z = ind.z + dz;
    /* ... */
}
```

同时本文还使用了 NVCC 编译器提供的循环展开指令#pragma unroll，将邻居查找循环在每个维度上展开三次。在 GPU 指令层面，此时遍历相邻格子的过程彻底没有了分支语句，进一步提高了性能。

**核函数内存优化：**CUDA 除了为核函数的每个线程提供了随机读写的全局显存，还对一个 Block 内的线程提供了一个很小的共享内存（Shared memory）。共享内存从 GPU 的 L1 缓存分割而来，大小在调用核函数时指定，通常不会超过 48KB 大小。因为 L1 缓存的速度比全局显存快得多，设计程序正确地使用共享内存，能够带来很大的性能。图 2.12 展示了 CUDA 核心的内存结构。

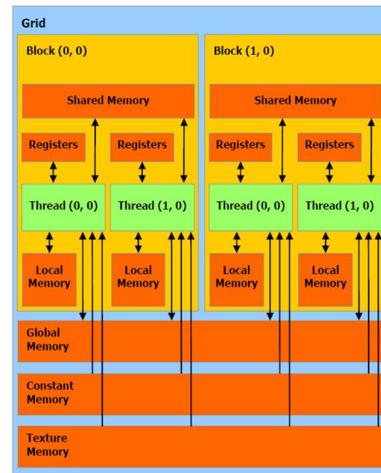


图 2.12 CUDA 核心内存结构[43]

共享函数一个典型应用是做手动的、可配置的高速缓存。若预先知道全局显存中有数据会被 Block 内不同线程读取，本文可以预先将数据从全局缓存中挪到共享内存中，此后不同线程从共享内存中读取即可。在 `computeGridRange` 核函数中，每个线程读取了当前粒子的格子编号和上一个粒子的格子编号，因此每个粒子的格子编号将会被读取两次（最开始和最后一个粒子除外）。本文预先将这个数据保存在共享内存中，减小了几乎一半的全局显存读取开销。

装

订

线

### 3 拉式流体的屏幕空间渲染

本章叙述将拉式流体还原为视觉真实的图像的渲染方法。本文着重研究流体中液体的渲染。朴素的 SPH 模型不能表现液体的平滑表面，因此并不具有视觉真实性。在屏幕空间（Screen space）的流体渲染技术[26][24]发表以前，液体表面的还原大多在世界空间（World space）内进行。例如，水平集（Level-sets）可以追踪液体表面的位置和拓扑变化[44]；移动立方体（Marching cubes）法从体素中还原多边形网格（Polygonal mesh）[22]。世界空间方法能得到精度高，视觉真实性较强的结果，但同时需要较大的计算量，因此不适合在实时应用中使用。本文在[24]的框架下，使用双边滤波器作为液体表面平滑的手段，提出了一种新的屏幕空间液体渲染方法。该方法仅利用距离摄像机最近一侧的粒子还原液体表面，不需要重建多边形网格，具有良好的性能和优越的实时性。本章 3.1 节叙述该算法的基本流程和框架，3.2 节介绍和推导本算法多处用到的 OpenGL 坐标变换和还原公式，3.3 节介绍算法中使用的液体深度纹理和厚度纹理的渲染，3.4 节介绍从深度纹理中还原液体表面法向量的方法，3.5 节叙述本文使用的液体深度纹理平滑方法，3.6 节叙述利用深度和厚度信息对液体表面进行着色的方法。

#### 3.1 算法流程概述

本渲染算法是屏幕空间内的渲染算法。算法首先将粒子坐标转换到摄像机空间，然后使用点状贴图（Point Sprite，或称点状精灵图）将粒子的深度和厚度渲染到屏幕空间的一张二维纹理中。深度信息记录了摄像机投影面上某像素到最近的粒子的距离，厚度信息记录了液体的厚度，这两个概念将在 3.2 节中详细解释。然后，为了得到平滑的液体表面，使用双边滤波器对深度纹理进行平滑操作。第三步从深度纹理中还原液体表面法向量，并将法向量存在一张二维纹理中。最后结合液体折射的菲涅尔法则，光线追踪算法和天空盒模型，对液体进行着色，渲染出最终的液体画面。自第一步得到两张屏幕空间纹理，后续的算法步骤都在这两张纹理上进行，不再用到原始的粒子位置，因此称此方法为拉式液体的屏幕空间渲染方法。算法流程如图 3.1 所示：

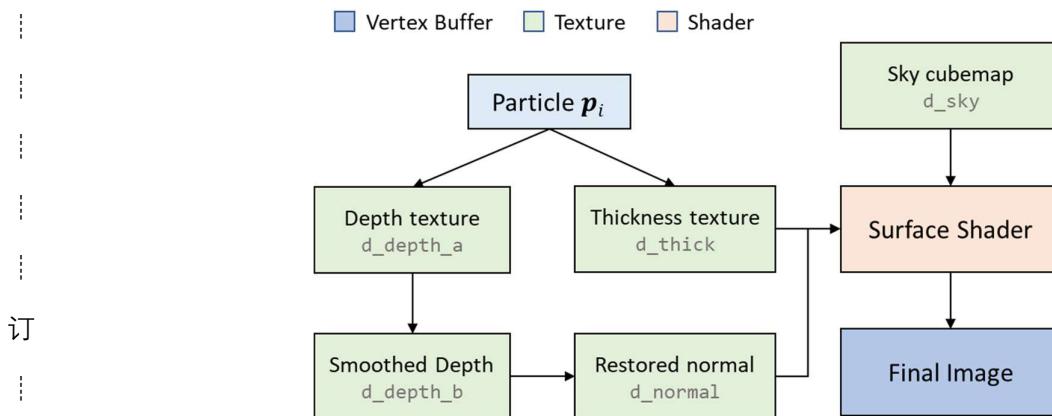


图 3.1 渲染算法流程

渲染算法全部使用 OpenGL 4.6 GLSL 语言（OpenGL Shading Language）完成。关键的计算步骤，如液体深度平滑和表面法向量重建，也可以使用 OpenGL compute shader 或 CUDA 完成，它们能提供更高的自由度。但是仅就本文着眼的屏幕空间液体渲染而言，它们引入了不必要的复杂性。因此在本文的实现中计算过程全部是使用 OpenGL 片元着色器（Fragment shader）完成的。这些用于计算的着色器以纹理为输入，渲染到后台的 framebuffer 上。Framebuffer 是 OpenGL 片元着色器的渲染目标，通常是一个二维的纹理，DirectX 中对应的概念是 rendertarget。OpenGL 有一个默认的 Framebuffer，绑定到屏幕输出。手动在 OpenGL 中申请并绑定 framebuffer，则可以在后台完成诸如图像后处理的像素级并行任务。

以下列出渲染算法使用的所有纹理以及他们的类型。这些纹理都绑定在名为 `d_fbo` 的 Framebuffer 上。需要注意的是，OpenGL 设计之初是用于 3D 渲染的图形 API，因此纹理的数据类型都与颜色、深度等图形概念有关。例如，本文需要使用 `GL_R32F` 类型存储 32 位精度的纹理，使用 `GL_RGB32F` 存储向量纹理。这不妨碍本文将其视作普通的 `float` 和 `float3` 类型使用。

表 3.1 渲染算法使用的纹理

名称	维度	类型	描述
<code>d_depth</code>	<code>GL_TEXTURE_2D</code>	<code>GL_DEPTH_COMPONENT32F</code>	用于深度监测的深度纹理*
<code>d_depth_a</code>	<code>GL_TEXTURE_2D</code>	<code>GL_R32F</code>	摄像机空间深度纹理*
<code>d_depth_b</code>	<code>GL_TEXTURE_2D</code>	<code>GL_R32F</code>	备用摄像机空间深度纹理**
<code>d_normal</code>	<code>GL_TEXTURE_2D</code>	<code>GL_RGB32F</code>	法向量纹理
<code>d_thick</code>	<code>GL_TEXTURE_2D</code>	<code>GL_R32F</code>	厚度纹理
<code>d_sky</code>	<code>GL_TEXTURE_CUBE_MAP</code>	<code>GL_RGB</code>	天空盒贴图

\* `d_depth` 用于启用 OpenGL 深度监测（Depth test），以正确的渲染物体的遮挡关系，类型必须为 `GL_DEPTH_COMPONENT` 或其子类型，且不能作为着色器的输入，因此不能替代 `d_depth_a`。

\*\* 使用两套深度纹理（双重缓冲技术），避免在深度平滑步骤中不必要的来回拷贝。

装

以下按照渲染 Pass（步骤）顺序列出算法所使用的着色器，以及每个着色器的输入与输出。每个着色器由一个顶点着色器（Vertex Shader）和一个片元着色器（Fragment Shader）组成。计算基本上在片元着色器上完成。除 `get_depth` 和 `get_thick` 着色器的输入是顶点缓存外，其他三个着色器的输入输出都是纹理。

表 3.2 渲染算法使用的着色器

着色器名称	输入	输出	描述
<code>get_depth</code>	<code>vertex_vao</code>	<code>d_depth, d_depth_a</code>	用于深度监测的深度纹理*
<code>get_thick</code>	<code>vertex_vao</code>	<code>d_thick</code>	摄像机空间深度纹理*
<code>smooth_depth</code>	<code>d_depth_a*</code>	<code>d_depth_b*</code>	备用摄像机空间深度纹理**
<code>restore_normal</code>	<code>d_depth_b*</code>	<code>d_normal</code>	法向量纹理

订

线

续表 3.2

shading	d_normal, d_thick, d_sky	screen**	厚度纹理
---------	-----------------------------	----------	------

\*深度纹理使用了多重缓冲技术。此处用  $d\_depth\_a$  代表上一轮结果， $d\_depth\_b$  代表下一轮写入目标。 $restore\_normal$  的输入是深度的最终平滑结果。

\*\*绘制到屏幕上，屏幕纹理是 OpenGL 的默认输出纹理。

### 3.2 渲染管线的坐标变换[45]

本节介绍 OpenGL 渲染管线中顶点坐标转换的概念。粒子坐标进入管线时位于世界坐标系。经由多步变换，分别变换到摄像机坐标系(Eye Coordinates)，裁剪坐标系（Clip coordinates），归一化设备坐标系（Normalized device coordinates, NDC），最后到屏幕坐标系(Window Coordinates)。屏幕空间渲染算法的一个关键步骤，是利用二维纹理的坐标和深度还原其对应片元在摄像机坐标系中的坐标。因此搞清楚 OpenGL 各步变换和逆变换是实现屏幕空间渲染算法的必要条件。此处，屏幕空间是一个与世界空间相对的概念。屏幕空间渲染泛指利用二维纹理在摄像机空间、裁剪坐标系和 NDC 坐标系做渲染的做法，与屏幕坐标系是不同的概念。

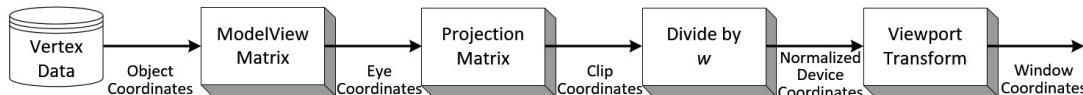


图 3.2 OpenGL 顶点坐标变换过程[46]

为了统一平移和旋转、缩放变换，OpenGL 使用齐次坐标系记录顶点的坐标。齐次坐标系有四个维度，包括三个位置维度和一个齐次维度（Homogeneous coordinate）。齐次坐标为 0 表示向量，不为零表示空间中的一个点。在齐次坐标系下，OpenGL 中的任何变换都可统一地表示成  $4 \times 4$  维的矩阵。关于齐次坐标系的设计和原理不是本文的重点，在此不做展开。

顶点坐标从世界坐标系到裁剪坐标系的过程是可以在顶点着色器中控制的。在顶点着色器中，本文将顶点坐标左乘 ModelView 变换矩阵得到摄像机空间坐标。ModelView 矩阵通常是一个具有平移、旋转和缩放效果的矩阵。接着，本文将摄像机空间坐标左乘投影变换矩阵，得到裁剪空间坐标，将这个坐标赋给顶点着色器内置变量  $gl\_Position$ ，之后的变换由 OpenGL 自动完成。

投影矩阵将顶点从摄像机空间变换到裁剪空间。OpenGL 通过简单的把裁剪空间中的坐标除以齐次坐标得到 NDC 坐标，因此实际上投影矩阵包含了从摄像机坐标变换到 NDC 坐标的。本文的渲染算法需要将坐标从 NDC 空间还原到摄像机空间，因此投影变换矩阵是本文研究的重点。

本文使用投影是符合人眼视觉规律的透视投影。透视投影矩阵将摄像机空间内一个平截头体（Frustum）性地变换到一个立方体 NDC 空间中。因为这个平截头体摆在假象的摄像机前，又将其称为视体（Viewing frustum）。视体的形状、位置和大小由摄像机坐标下的  $l, r, t, b, n, f$  六个

参数确定，分别的代表视体的左右上下边界和前后截面的z轴坐标。注意在 OpenGL 使用的右手坐标系中，整个视体都位于z轴的负半轴区域内，因此前截面的z坐标为 $-n$ ，后截面的z坐标为 $-f$ 。

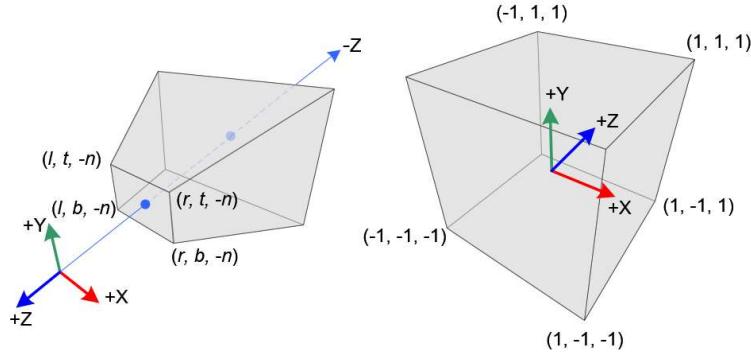


图 3.3 投影变换的平截头体[46]

透视投影可以分解成两个易于理解和计算的步骤。对视体中的任意一点，首先将其投影到前截面上，再将前截面缩放成长宽范围是 $[-1,1]$ 的正方形。图 3.4 以 $x$ 坐标为例，展示了视体中一点投影到前截面的过程。

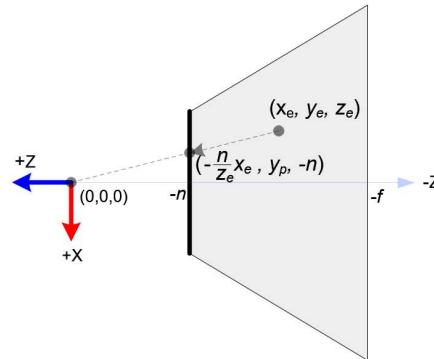


图 3.4 透视变换 $x$ 坐标投影示意图[46]

其中摄像机空间的原点是摄像机所在位置。设顶点摄像机空间坐标为 $(x_e, y_e, z_e)$ ，投影到前截面后坐标为 $(x_p, y_p, z_p)$ ，由相似关系容易得到

$$x_p = -\frac{n}{z_e} x_e \quad (3.1)$$

$$y_p = -\frac{n}{z_e} y_e \quad (3.2)$$

再将前截面缩放成长宽范围是 $[-1,1]$ 的正方形。缩放后得到的是顶点的 NDC 坐标，记为 $(x_n, y_n, z_n)$ ，有

装

订

线

$$x_n = \frac{2x_p}{r-l} - \frac{r+l}{r-l} \quad (3.3)$$

$$y_n = \frac{2y_p}{r-l} - \frac{r+l}{r-l} \quad (3.4)$$

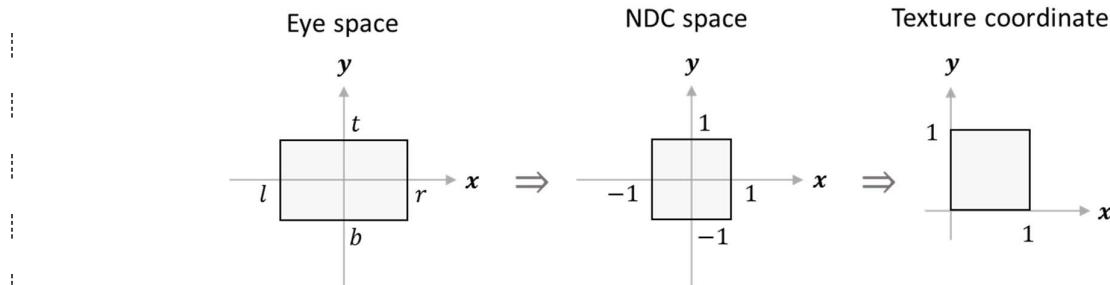


图 3.5 视体前截面转换到 NDC 坐标系、贴图坐标系

通常本文希望视体中心对称，此时有  $r = -l, t = -b$ 。在此条件下，综合(3.1)到(3.4)，本文给出从 NDC 坐标系还原摄像机坐标系的公式

$$x_e = -\frac{n}{z_e r} x_n \quad (3.5)$$

$$y_e = -\frac{n}{z_e r} y_n \quad (3.6)$$

贴图纹理进一步将 NDC 坐标系映射为  $xy$  范围在  $[0,1]$  的正方形。类似的，设顶点贴图纹理为  $(x_t, y_t)$ ，摄像机空间  $z$  坐标为  $z_e$ ，则可还原得到摄像机空间  $xy$  坐标。

$$x_e = -\frac{n}{z_e r} (2x_t - 1) \quad (3.7)$$

$$y_e = -\frac{n}{z_e r} (2y_t - 1) \quad (3.8)$$

在后面几节中，本文将利用公式 (3.5) 到公式 (3.8) 还原纹理像素的摄像机空间坐标。

### 3.3 深度渲染和厚度渲染

屏幕空间液体渲染的第一步，是将液体的深度和厚度信息投影到摄像机平面上，并分别使用两张二维纹理存储。每张二维纹理的分辨率与渲染画面的分辨率相同，上面的每个像素使用一个 32 位浮点数存储信息。此后，本文对液体表面的平滑、还原和着色都在这两张纹理上进行。

屏幕空间液体渲染巧妙之处在于实现了从二维纹理到表面法向量的还原。本文知道，所有光照模型的着色计算都需要表面法向量的参与，因此这种方法能够表现出复杂的表面光照特性。同时由于算法将液体粒子的表面重建转化为二维纹理的平滑问题，本文得以应用计算机视觉领域里各种成熟的平滑技术，如 3.4 节介绍的本文使用的双边滤波算法。

本文首先定义深度的概念。若没有特别说明，本文所指的深度，是摄像机坐标系中二维纹

理记录的 z 轴坐标。这个深度存储在 `d_depth_a` 和 `d_depth_b` 两张纹理上，称为深度纹理（Depth texture）。同时，另有一个深度监测纹理（Depth test buffer），用于正确的渲染粒子的遮挡关系。

下面本文详细描述深度纹理的渲染方法。这种方法使用 OpenGL 点模式绘制粒子，利用 glsl 语言控制点的大小、形状和输出结果，被称为 Point sprites 方法。在深度渲染 Pass 中，本文使用 OpenGL 的 GL\_POINTS 模式向 `d_depth_a` 绘制所有粒子。GL\_POINTS 会将顶点绘制为一块正方形（quad），正方形的大小通过顶点着色器内置变量 `gl_PointSize` 指定。为了将粒子绘制为圆形，可以在片元着色器中使用 `discard` 命令，将正方形中超出内接圆的像素扔弃。为了向深度纹理写入深度信息，本文在片元着色器中计算好当前像素点的 z 坐标并返回到输出即可。

由于本文独立地绘制顶点，而不是绘制多边形，透视投影的效果无法体现。因此本文需要手动地根据粒子距离摄像机的远近指定粒子的大小，以保证粒子正确的透视关系。假设粒子在世界坐标系中的半径为  $R$ ，用类似公式(3.5)的推导过程，本文可以得到正确的 quad 大小  $S_q$

$$S_q = S_h R \frac{n}{z_e t} \quad (3.9)$$

其中  $n, t, z_e$  的定义同公式 (3.5)， $S_h$  是屏幕宽度，单位为像素。

确定了正确的粒子大小后，还需要把光栅化得到的正方形裁剪成圆形。OpenGL 在片元着色器中提供了内置变量 `gl_PointCoord`，类型为 `vec2`，记录了当前片元在正方形中的坐标。坐标以正方形的左上角为顶点，范围是  $[0,1]$ 。本文可以利用 `gl_PointCoord` 将圆形外的部分剔除，如图 3.6 所示。

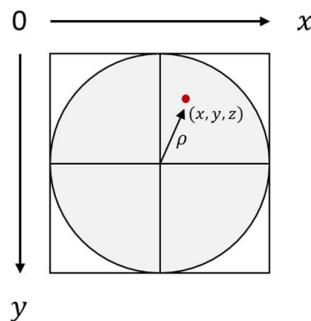


图 3.6 在 quad 中还原球面坐标

舍弃  $\rho = \sqrt{x^2 + y^2} > 1$  的像素即可将粒子还原成圆形。同时已知粒子中心深度  $z_e$ ，粒子大小  $R$  和像素坐标  $(x, y, z)$ ，由初等立体几何可以得到该像素点的深度  $z_d = z_e + zR$ 。将  $z_d$  从片元着色器中输出即可写入深度缓存中。

需要注意的是，在进行深度渲染时需要开启 OpenGL 的深度测试。此时 OpenGL 会根据向深度测试缓存写入像素的深度信息。当新的像素从片元着色器输出时，OpenGL 会先判断当前像素

装

订

线

和深度测试缓存中的深度大小<sup>1</sup>[45]。若当前像素的深度较深，则丢弃当前像素，否则使用当前像素覆盖旧的数据。开启深度测试使得本文只渲染了最靠近摄像机一侧的液体表面。考虑到液体是透明的，光线可能会从光源出发，在液体和气体间经过多次，才达到摄像机（例如 LSSSS 路径，液体穿过了两块液体，共四个 Specular 表面，最终进入摄像机），屏幕空间渲染无法捕捉这样的着色信息。这样复杂的光路即使离线渲染也很难得到良好的效果。对于最普遍的，光线只经过了一块液体的情形，本文认为屏幕空间渲染是可以胜任的。

下面本文叙述厚度渲染的方法。厚度（Thickness）表示从摄像机看来某个位置液体的厚度。根据比尔-朗博定律（Beer–Lambert law），液体的透光量随液体的厚度呈指数衰减。根据液体厚度应用不同的着色，有助于表现液体的空间层次感。厚度纹理与深度纹理具有一样的大小和类型。不同的是，渲染厚度纹理需要关闭深度测试，保证前后遮挡的粒子对厚度有通常的贡献。同时，深度纹理的混合模式（Blending mode）设置为等权重叠加模式。圆形粒子的厚度特征是中心厚，四周薄。为了方便计算，本文没有严格地计算球体厚度，而是受 Phong shading 启发使用了一个近似的公式。设球面上坐标为  $(x, y, z)$  的像素厚度为  $T$ ，球的半径为  $R$ ，有  $T = R(x, y, z) \cdot (0, 0, 1) = zR$ 。



图 3.7 深度纹理和厚度纹理。  
使用了修改的 sigmoid 函数重映射值域到  $[0,1]$

### 3.3 表面法向量重建

本节本文介绍从深度纹理还原液体表面法向量的方法。对深度纹理上一点  $(s, t)$ ，若这个位置上有液体，设它在摄像机坐标系中的坐标为  $\mathbf{P}(s, t) = (x, y, z)^T$ ，则  $z$  可以从深度纹理中读取， $x$  和  $y$  则可以利用公式 (3.7) 和 (3.8) 得到。由微分几何知识，此处液体表面的法向量  $\mathbf{n}(s, t)$  可

<sup>1</sup> 深度测试也有可能紧接着顶点着色之后执行（Early Fragment Test）。但由于本文改写了片元着色器的 gl\_FragDepth，这种情况不会发生。

从以下公式得出

$$\mathbf{n}(s, t) = \frac{\partial \mathbf{P}}{\partial s} \times \frac{\partial \mathbf{P}}{\partial t} \quad (3.10)$$

可以用偏微分的一阶差分近似求公式 (3.10)

$$\frac{\partial \mathbf{P}}{\partial s} \approx \frac{\mathbf{P}(s + \Delta s, t) - \mathbf{P}(s, t)}{\Delta s} \quad (3.11)$$

$$\frac{\partial \mathbf{P}}{\partial t} \approx \frac{\mathbf{P}(s, t + \Delta t) - \mathbf{P}(s, t)}{\Delta t} \quad (3.12)$$

其中， $\Delta s$  和  $\Delta t$  分别表示横向和纵向上一个像素宽度对应的纹理左边偏移值。综合 (3.10) , (3.11) , (3.12) , (3.7) 和 (3.8) , 可以将公式 (3.12) 进一步改写成以像素坐标( $u, v$ )表示的形式。

$$\mathbf{n} = \begin{pmatrix} -C_y \Delta_u z \\ -C_x \Delta_v z \\ C_x C_y z \end{pmatrix}$$

$$C_x = \frac{2r}{S_w n}, \quad C_y = \frac{2t}{S_h n} \quad (3.13)$$

$$\Delta_u z = \min(\mathbf{P}(u - 1, v), \mathbf{P}(u + 1, v)), \quad \Delta_v z = \min(\mathbf{P}(u, v - 1), \mathbf{P}(u, v + 1))$$

需要注意的是，OpenGL 提供的像素坐标纹理 API 将纹理值记录在整点网格的中心。例如，纹理左下角第一个像素的值可通过坐标(0.5,0.5)取到。同时，本文将纹理的采样模式设为线性插值 (Linear Interpolation, OpenGL 中 GL\_LINEAR 标志位) , 这可以为本文提供微小的深度平滑效果。

由于公式 (3.10) 中  $\mathbf{P}$  在粒子的边缘处不连续，导致  $\mathbf{P}$  不可微，此公式会失效。图 3.8 使用红绿蓝分别表示法向量三个维度的绝对值。若不做处理，边缘处的法向量可能很小，造成图 3.8 左侧的边缘效果。作为改进，在边缘处使用  $\mathbf{P}$  连续一侧的偏导数。体现在公式上，则是取前后差分中较小的一个。

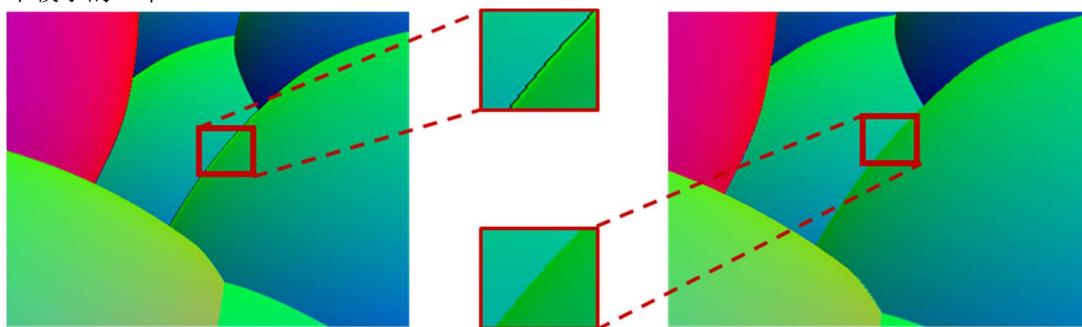


图 3.8 边缘处的深度差分。左：正确边缘；右：错误边缘。

订

线

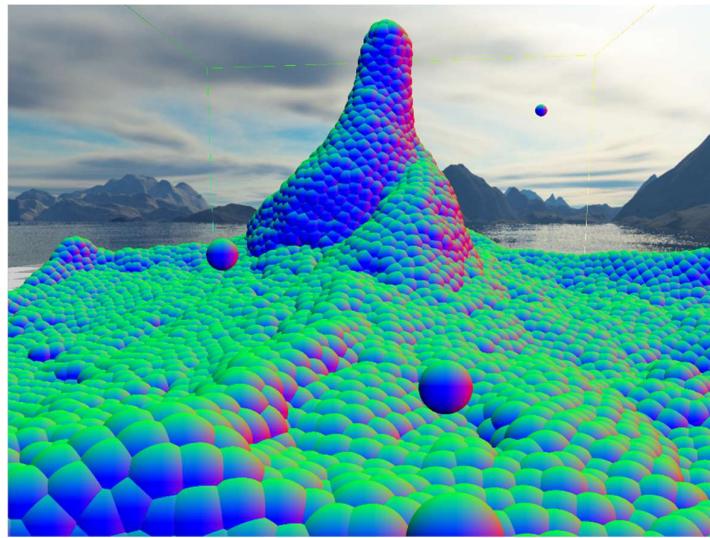


图 3.9 从深度纹理重建表面法向量

### 3.4 深度平滑

从图 3.9 可以看出未经平滑的液体表面有明显的粒子分界。本文希望液体表面能够尽量平滑，隐藏其粒子模拟的本质。

一种可行方法是，直接使用二维图像的平滑算法对法向量纹理进行平滑。由于深度纹理没有发生改变，还原的像素位置也不会发生改变。但由于表面法向量的改变蕴含着像素的空间位置的改变，导致这种方法在进行反射、折射现象的着色时，光线追踪的出发点存在误差，因此并不能精确的反应着色的结果。

为了克服这个困难，本文直接对深度纹理进行平滑，再从平滑后的深度纹理还原法向量。由于深度纹理是一张灰度图，更有利于本文使用计算机视觉领域中丰富的灰度图平滑算法。本文使用双边滤波算法作为平滑算法。双边滤波算法是高斯模糊算法的一个改进，它们将图像中的一个像素替换称为它邻居像素的加权和，以达到平滑的效果。高斯模糊算法中，邻居像素的权重仅考虑它与中心像素的距离。而双边滤波算法则在此基础上综合考虑了它与中心像素的深度差。距离越远，深度差越大，像素的权重越小。

对像素 $x$ ，设其深度为 $I(x)$ ，经双边滤波算法处理后的深度 $I^{\text{filter}}$ 由以下公式给出

$$I^{\text{filter}}(x) = \frac{1}{W_p} \sum_{y \in \Omega} I(y) f_r(\|I(y) - I(x)\|) g_s(\|x - y\|) \quad (3.14)$$

$$W_p = \sum_{y \in \Omega} f_r(\|I(y) - I(x)\|) g_s(\|x - y\|) \quad (3.15)$$

$$f_r(d) = e^{-\sigma_r d}, \quad g_s(d) = e^{-\sigma_s d} \quad (3.16)$$

订

线

其中， $\Omega$ 是采样区域，通常是一个以 $x$ 为中心的圆， $W_p$ 是权重归一化系数， $\sigma_r$ 和 $\sigma_d$ 分别是距离模糊半径和深度模糊半径， $f_r$ 和 $g_s$ 分别是距离和深度模糊权重函数。若为高斯模糊，不考虑像素间深度差。表现在公式上，则为 $f_r \equiv 1$ 。

双边滤波算法的特点是它可以在平滑的同时保留图像中的边缘。在本文的深度图中，边缘有两种情况，一种是液体与非液体的边缘，一种是液体与液体，但不在一个连续深度范围的边缘。后者对表现液体的深度关系非常重要。如果使用高斯模糊算法，这样的边缘会被模糊掉，本来不连续的两个深度上液体将会错误地被渲染成一块。

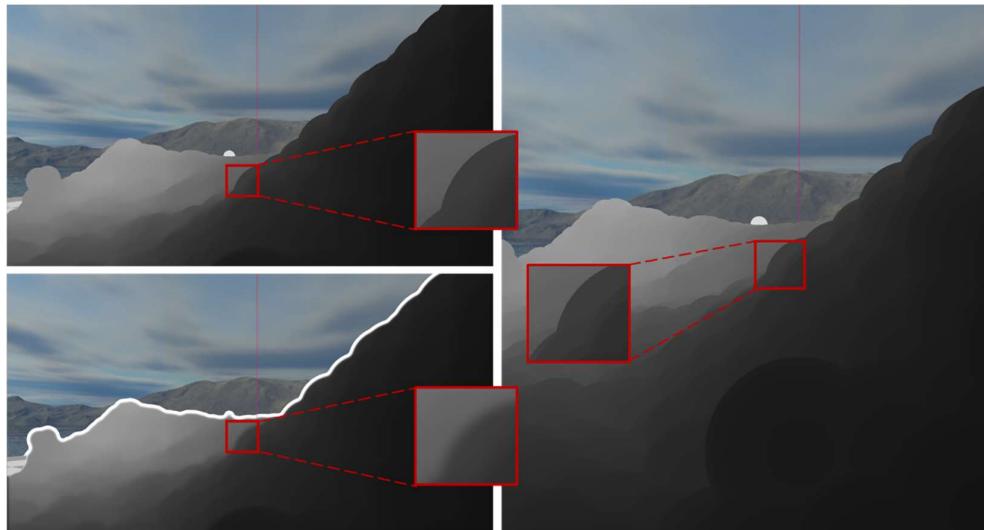


图 3.10 平滑前后深度图。左上：双边滤波，左下：高斯模糊，右：未平滑

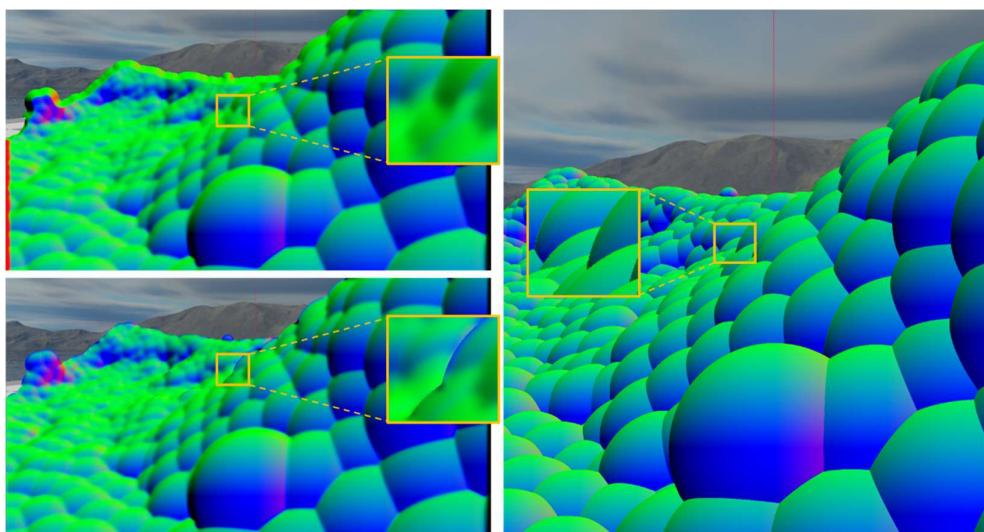


图 3.11 平滑前后法向量图。左上：高斯模糊，左下：双边滤波，右：未平滑

装

订

线

上图是平滑前后的深度图和法向量图。本文标注的区域含有一条边缘，边缘两侧液体有明显的深度差。注意到，高斯模糊算法不能保持边缘，导致平滑后边缘被弱化，形成错误的液体表面层次关系。同时在整个图像的左侧边缘，高斯模糊出现了红色法向量的伪像。双边滤波算法则没有这个问题。

本文使用平滑着色器反复迭代多次，交替的以 `d_depth_a` 和 `d_depth_b` 作为输入和输出（双重缓冲法）。由于双边滤波是一个  $O(n^2)$  复杂度的算法， $n$  是采样半径，本文不能使用太大的  $n$ 。在实现中，本文取两次迭代，采样半径为 10 个像素。考虑到多次平滑有传递采样信息的效果，这种方法可以以相对较小的代价达到与大采样半径近似的效果。

### 3.5 表面着色

得到液体表面的法向量信息和厚度信息后，本文对液体表面进行着色操作，并输出最终的液体图像。液体表面着色不单单要考虑液体本身的信息，也要考虑液体外部的物体和光照信息。特别是考虑到液体表现出半透明的性质，正确的处理液体表面反射/折射的光线来源，是渲染具有真实感的液体的必要条件。除了渲染液体外，本文还需要渲染外部的天空盒和地平面。整个渲染管线图 3.12 所示

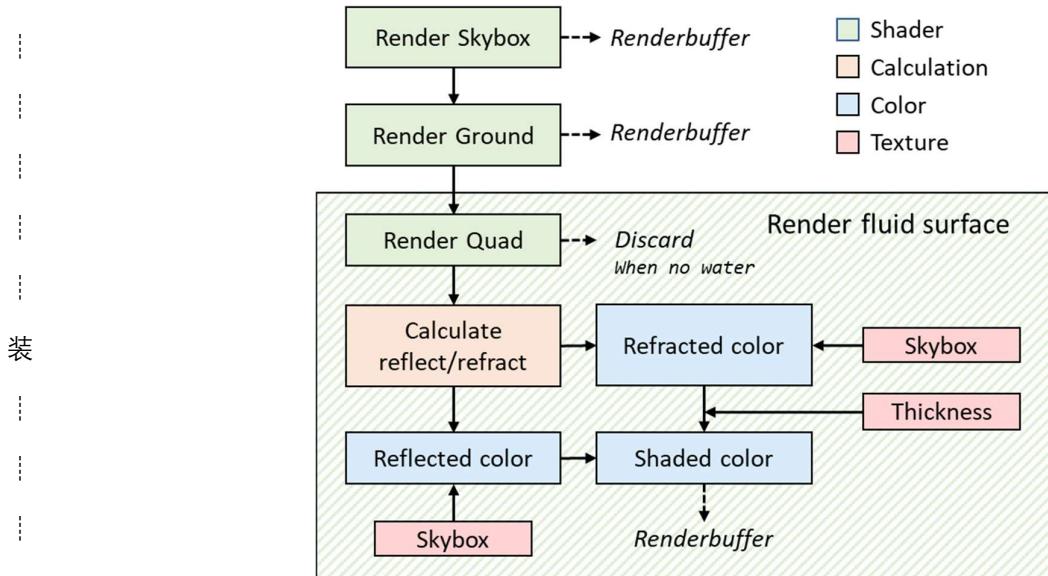


图 3.12 渲染管线流程

首先本文向 Renderbuffer 渲染天空盒和棋盘格地面。然后，本文渲染一张覆盖全屏幕大小的长方形（称为 quad，由两个三角形组成），同时挂载深度测试纹理、深度纹理、法向量纹理和

厚度纹理。在片元着色器中，本文读取深度测试纹理中的深度信息<sup>2</sup>，将其赋给内置变量 `gl_FragDepth`。这将使得 OpenGL 丢弃被遮挡的液体。没有被丢弃的像素则是液体表面的像素。利用公式（3.7），（3.8）得到摄像机空间位置，可以算出从此处入射摄像机的光线方向。又从法向量纹理中取得了法向量，本文可以得到此处光线的反射、折射情况。光路如图 3.13 所示

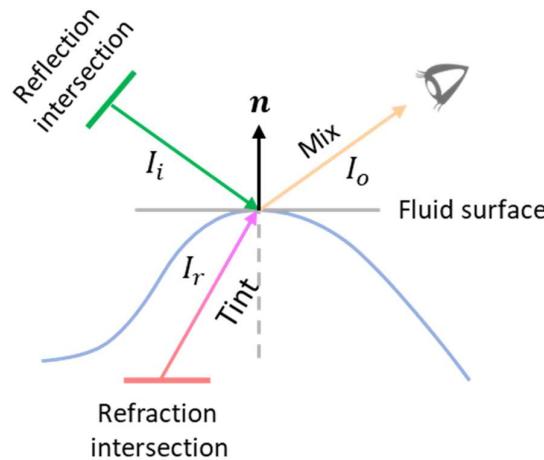


图 3.13 液体表面光路图

菲涅尔方程给出了光线在水等电解质表面反射和折射时光线曲折和能量分布的规律。菲涅尔方程的精确形式较为复杂，且根据入射光极性的不同，能量的分布也有不同的形式。为了节省计算资源，计算机图形学常常使用菲涅尔方程的 Schlick 近似。Schlick 近似给出了反射系数  $R$ ，表示反射光占入射光的能量的比例。设入射光与法向量的夹角为  $\theta$ ，液体和空气的折射率分别为  $n_1$  和  $n_2$ ，Schlick 近似由下面的公式给出

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos \theta)^5 \quad (3.17)$$

$$R_0 = \left( \frac{n_1 - n_2}{n_1 + n_2} \right)^2 \quad (3.18)$$

Schlick 近似在  $\theta$  较小时有较好的近似效果。当  $\theta$  增大时，Schlick 近似开始出现比较大的误差。特别需要注意的是，Schlick 近似不能反应液体的全反射现象。可以利用全反射角计算公式  $\theta_c = \arcsin \frac{n_1}{n_2}$  特殊判断全反射。由于本文的应用都是从外部观察液体，因此不存在全反射的问题。在实际应用中，为了更强地表现液体表面的光泽，常常将公式（3.17）中  $(1 - \cos \theta)$  项的次数降低。本文实现中将次数降到了 3。

<sup>2</sup> 也可以只挂载深度纹理，并手动从摄像机空间将深度映射到深度测试纹理的深度。理论上 OpenGL 4.6 尚不支持直接读取深度测试纹理中的数据，因此本文就是用该方法实现的。具体映射方法本文不做展开。

反射系数 $R$ 表示了出射光中来自反射部分的比例。为了计算出射光，本文还需要知道反射光和折射光的颜色。反射光简单地通过反射定律给出。设出射光方向为 $\hat{I}_i$ ，反射光反向为 $\hat{I}_r$ ，都指向液体外部，有

$$\hat{I}_i = -\hat{I}_o + 2\mathbf{n} \quad (3.19)$$

为了计算折射光方向，本文不使用折射定律，而是将出射光地反向稍微向法线方向偏移。这个近似避免了折射定律中三角函数地计算。设折射光方向 $\hat{I}_r$ ，方向朝向液体内部，有

$$\hat{I}_r = -\hat{I}_o - c \cdot \mathbf{n} \quad (3.20)$$

本文取 $c = 0.2$ 能得到比较好的近似效果。出射光颜色 $I_o$ 由反射光 $I_i$ 和折射光混合得到 $I_r$ ，

$$I_o = RI_i + (1 - R)I_r \quad (3.21)$$

图 3.14 以灰度图的形式表现了反射系数 $R$ 在液体不同位置的取值

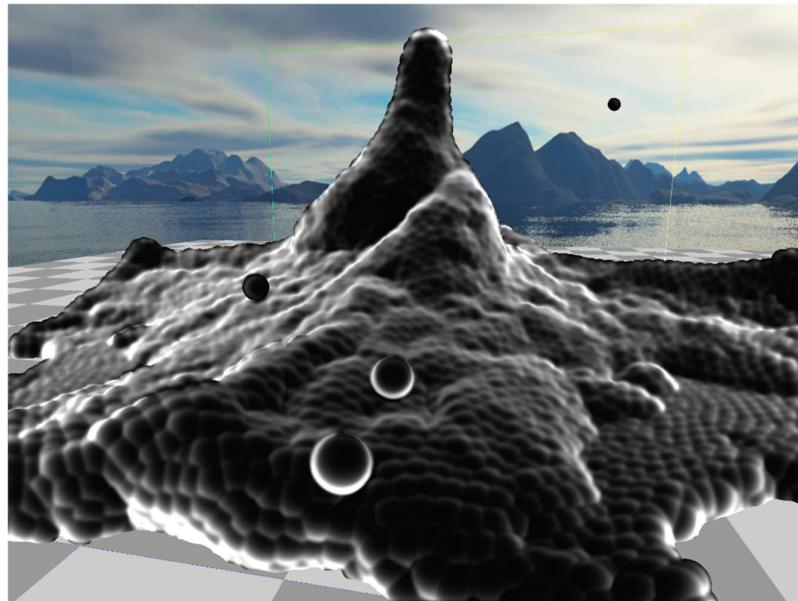


图 3.14 液体表面的反射系数。  
图中悬浮的粒子是独立于主水体外的孤立粒子

已知光线的起点和方向，可以通过光线追踪算法（Raytracing）获得反射光和折射光的颜色，追踪过程中本文忽视液体的存在，得到全局光照的单次光路近似。本文使用的外部场景仅包括一个棋盘格地面和一个天空盒，因此光线追踪算法不会消耗太多的计算时间。对追踪得到的折射光，还需要根据液体厚度进行染色（Tint）操作，使液体较厚的部分表现出深色的效果。根据 Beer–Lambert 定律，液体的透光量随液体的厚度呈指数衰减。本文在此给出修改的 Beer–Lambert 染色公式，使得液体不会因为太厚而完全不透光。设液体厚度为 $T$ ，追踪所得折射光线颜色 $I_r$ ，

装

订

线

染色后颜色为 $I_r'$ , 液体的固有颜色为 $I_f$ , 有

$$I_r' = AI_r + (1 - A)I_f \quad (3.22)$$

$$A = \max(e^{-0.5T}, 0.2) \quad (3.23)$$

综合以上着色公式和方法, 本文得到最终的渲染结果

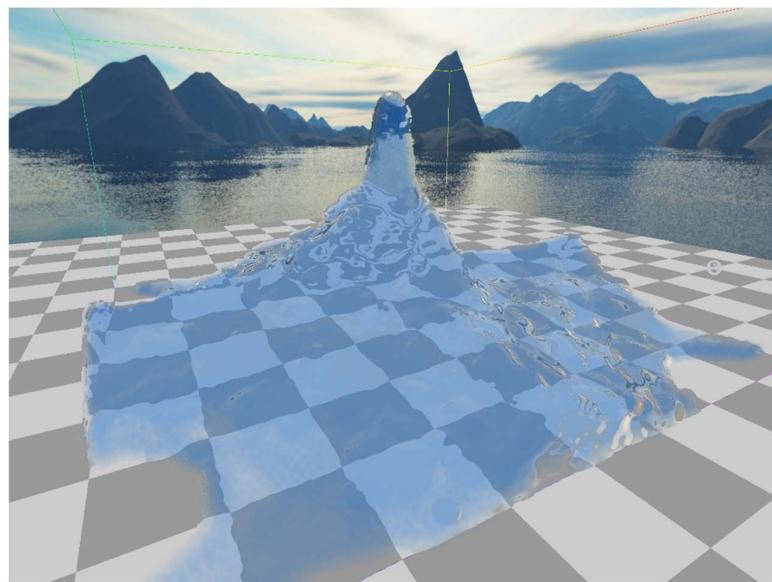


图 3.15 最终渲染结果

装

订

线

## 4 结果与讨论

本节分两个部分评估本文所做的工作。4.1 节展示了流体模拟文献中常见的三个测试场景。4.2 节测试了典型参数下系统各部分耗时分析系统的性能。

### 4.1 结果展示

本节给出算法在三个典型场景中的运行效果。三个场景所使用的参数相同，罗列如下

表 4.1 测试场景所用系统参数

模块	名称	数值	描述
共有	ulim	(2,2,4)	界定框顶端定点
	llim	(-2,-2,0)	界定框底端顶点
模拟	$h$	0.1	核函数半径
	$\rho_0$	8000	静止密度
	$g$	9.8	重力加速度
	$\epsilon_\lambda$	1000	公式(2.29)修正量
	$\Delta q$	0.03	公式(2.31)参考粒子间距
	$k_{\text{corr}}$	0.001	公式(2.31)虚拟压强系数
	$n_{\text{corr}}$	4	公式(2.31)虚拟压强次数
	$c_{\text{XSPH}}$	0.5	公式(2.33)粘度系数
	$n$	4	压强修正迭代次数
	$n_{\text{smooth}}$	2	深度纹理平滑迭代次数
渲染	$r$	0.05	摄像机空间内粒子渲染大小
	$R_s$	10	平滑采样半径
	$\sigma_r$	6	距离平滑半径
	$\sigma_z$	0.1	深度平滑半径
	$S_W$	2000	渲染宽度
	$S_H$	1500	渲染高度

#### 4.1.1 单立方液体

在单立方体场景中，粒子初始从界定框的一个角落无初速度的落下，最后达到平稳状态。图 4.1 展示了该场景从初始状态到最后稳定的中间过程。本场景含有 27K 个粒子，分布在长宽高分别为  $1.5m \times 1.5m \times 1.5m$  的立方体中，从  $2.5m$  的高处落下。

#### 4.1.2 双立方液体

双立方液体场景又被称为双重溃坝（Double Dam Break）。几乎所有的液体渲染文献都会将 Double Dam Break 作为展示场景。在这个场景中，两块立方型液体从界定框的两个角落无初速度落下。两块液体在容器底部相遇后溅起强烈的水花，形成惊涛骇浪的效果。

图 4.2 展示了该场景从初始状态到最后稳定的中间过程。本场景含有两块液体，每块液体含 16K 个粒子，分别分布在两块长宽高为  $1m \times 1m \times 2m$  的立方体中，从  $2m$  的高处落下。

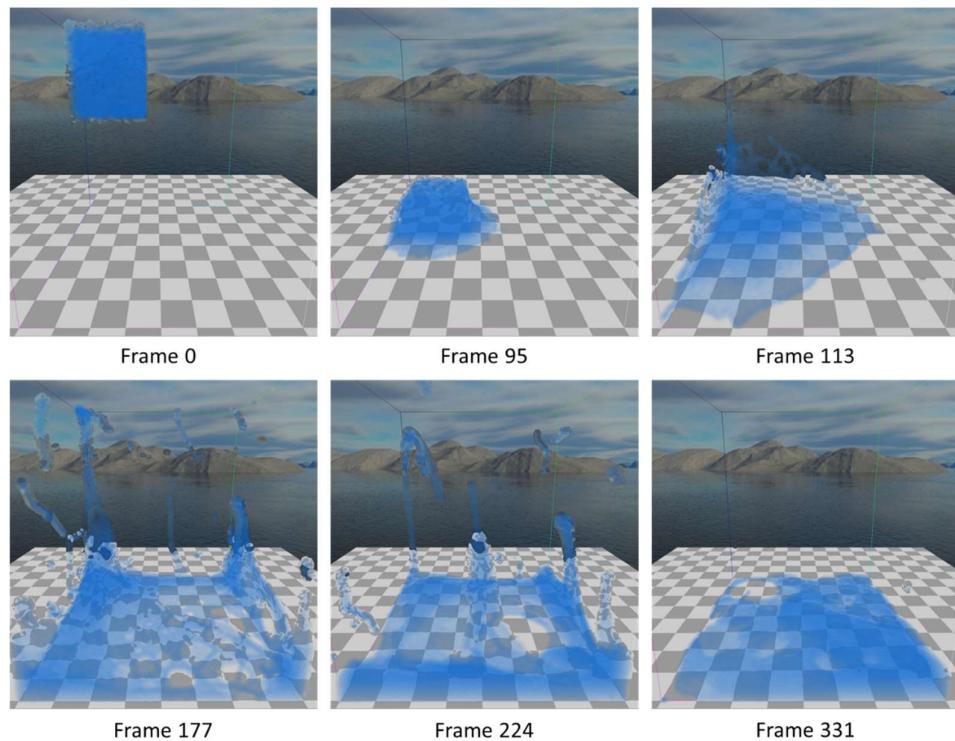


图 4.1 单立方液体场景

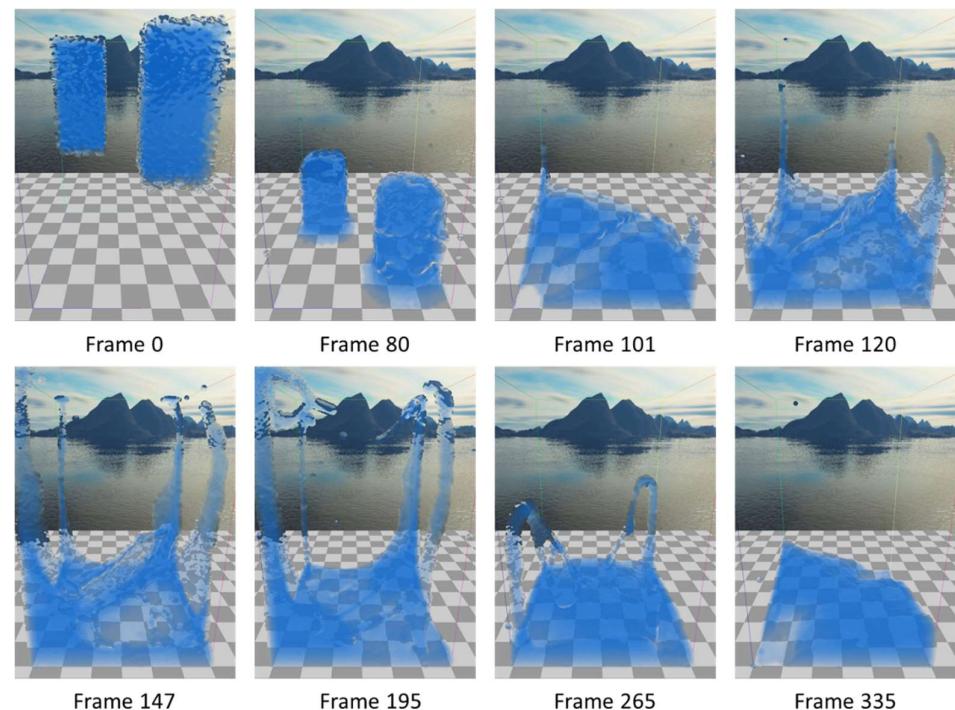


图 4.2 双立方体液体场景

装

订

线

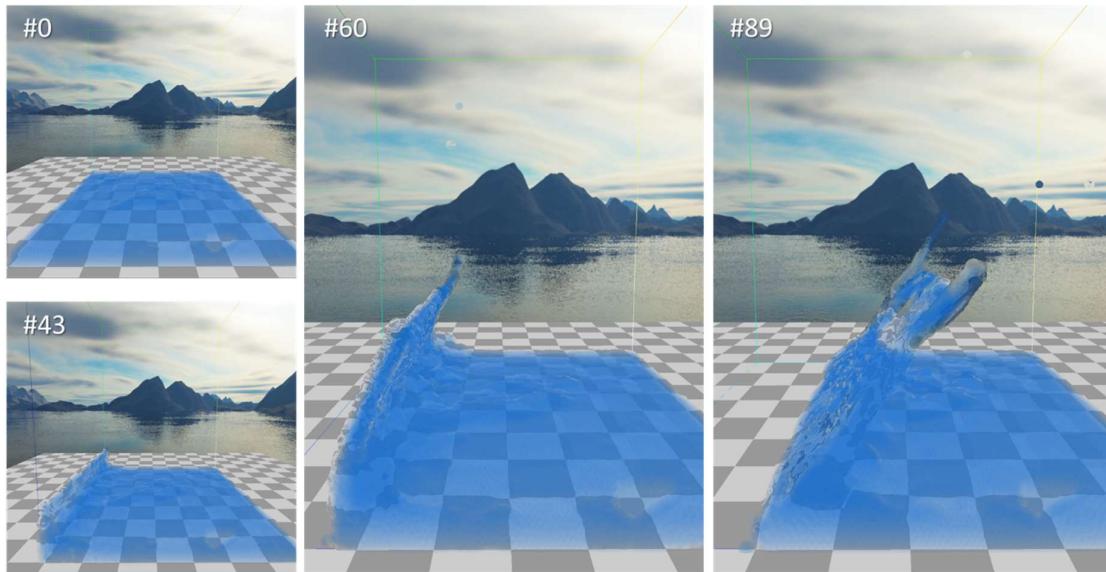


图 4.3 边缘移动场景

#### 4.1.3 边缘移动下的液体

在边缘移动场景中，界定框六个面中的一个面朝它的对面往复地进行靠近/远离正弦运动。初始时静止在容器底部的液体在界定框的压力下掀起波浪。图 4.3 展示了这个场景中波澜刚被掀起时的场景。本场景的初始状态是单立方液体场景平衡后的状态，含有 27K 个粒子。

以上三个实验中，粒子初始时等距地分布在立方体空间内，相邻两粒子间的距离恰好等于核函数半径。若距离大于或者小于核函数半径，会造成液体在空中膨胀或收缩，这不是本文所期望的。然而，如此严格等距的粒子之间不会有任何交互。当液体落到底部之后，也只能产生向上的反弹效果，而不能向四周散开。为了克服这个困难，本文对粒子初始位置加入了微小的扰动，扰动的幅度为核半径的  $1/10$ 。图 4.1 和图 4.2 的 0 帧中可以注意到这种扰动造成的液体表面微弱的涟漪。

本文简单分析算法运行的结果。模拟方面，本文的算法表现出了良好的真实感。首先注意到，液体能够正确处理与容器边界的碰撞。在容器的压力下，边缘液体密度增大，导致液体粒子被弹开，产生波浪效果。在双立方液体场景中，两块液体落到容器底部后在容器中心碰撞，形成对称的薄液壁，证明算法中没有系统误差，即算法是无偏的（Unbiased）。

渲染方面，厚度着色的应用给液体增加了丰富的层次感。例如 Double Dam Break 第 127 帧的容器底部，可以清晰看出液体的分布——经过碰撞后，液体击中在中部液壁的底部。折射光线和光线追踪的应用使得液体即使在完全静止的状态下也能表现出表面微小的起伏。例如移动边缘场景的第 0 帧，透过液体折射的棋盘格有轻微的扰动，这种扰动增强了液体表面的真实感。

## 4.2 性能分析

本节分两个情形评估本文工作的性能。本文的测试平台的配置参数如下：

CPU: Intel Core i7-7700HQ 2.8GHz

GPU: Nvidia GTX 1050 移动版

内存: 32G DDR4 1200MHz

CUDA 版本: CUDA 9.1

GPU 驱动版本: 391.35

操作系统: Windows 10 x64

编译器: MSVS C++ 14.1

### 4.2.1 粒子数量性能分析

本测试评估本文实现的模拟算法在不同粒子数量下的性能表现。本文使用的是屏幕空间渲染算法，渲染的耗时与渲染分辨率直接相关，而与粒子数量关系不大。因此本节我们只测试模拟算法，特别的，仅测试模拟算法中耗时最长的密度修正和哈希网格模块。测试的场景为双立方液体场景，通过控制液体的高度指定不同的粒子数量，同时保持长宽方向的粒子数量保持不变。测试从液体初始化开始，直至液体进入稳定状态结束。

表 4.2 给出了密度修正模块在一帧内消耗的平均时间，单位为毫秒，计时误差为 1.3 毫秒。可以发现，密度修正的时间随粒子数量的增大而增大，增大的速度略微超线性。这是由于我们使用的 GTX 1050 显卡只有 640 个核心，远远小于粒子数量。当我们开启与粒子等量的线程时，GPU 需要一定的存储空间和计算资源同时维持所有线程的状态。线程数量越多，线程切换的代价越大，同时缓存命中的几率也将降低。

作为对比，可以发现哈希网格的建立时间与粒子数量没有显著的关联。当粒子数量增大到 128K 时，算法仍然能以以小于 10 毫秒的速度执行，没有明显的性能退化。这是因为哈希网格最耗时的步骤 computeGridRange 核函数是以格子为并行单位执行的。在界定立方体和格子宽度未改变的情况下，格子数量也不会发生改变，因此该算法表现出与粒子数量无关的特性。实际上，该模块也应用了 GPU 上的基数排序算法，排序数组的长度等于粒子的数量。然而由于算法常数较小，128K 的粒子数量仍不能使其称为拖慢性能的决定性因素。

表 4.2 密度修正和哈希网格模块在不同粒子数量下的耗时统计

粒子数量	Density Correction	Hash Grid
8K	5.9	3.8
16K	13.2	1.8
32K	34.2	6.1
64K	71.7	4.4
128K	147.1	7.6

装

订

线

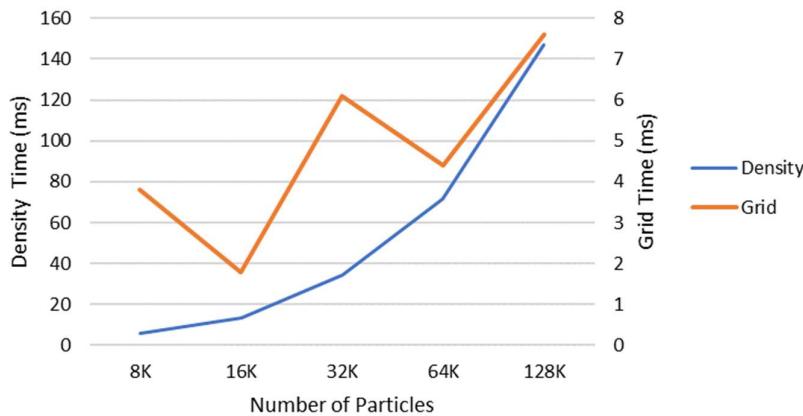


图 4.4 密度修正和哈希网格模块在不同粒子数量下的耗时统计

#### 4.2.2 模拟阶段性能分析

本测试评估算法在不同液体分布下的性能，测试场景为 4.1.2 节中的双立方液体。本测试将液体从初始状态到平衡状态分为四个阶段：第一个阶段，从第 0 帧到第 80 帧，称为平稳下落阶段，液体在下落过程中逐渐加速，但形状基本保持不变，粒子间作用力也基本保持平衡；第二个阶段，从第 81 帧到 265 帧，称为震荡阶段，液体粒子在这个阶段与容器壁和其他粒子发生激烈碰撞；第三个阶段，从第 265 帧到 335 帧，称为平衡恢复阶段，此时大部分液体位于容器底部，能量逐渐消耗，震荡幅度减弱。第四个阶段，从 335 帧到 400 帧，称为平衡阶段，本阶段液体基本处于平衡状态。

本测试统计模拟和渲染算法的各个模块在一帧内消耗的时间，平均值如表 4.3 所示，单位为毫秒，计时误差为 1.3 毫秒。

表 4.3 系统各模块耗时统计，单位：毫秒

	#1	#2	#3	#4
Density Correction	31.3	34.2	36.6	36.7
Hash Grid	4.7	4.5	3.5	3.8
Velocity Correction	2.8	2.8	3.0	3.0
Depth Rendering	0.7	0.7	0.1	0.1
Advection	0.3	0.2	0.2	0.2
Velocity Update	0.3	0.3	0.2	0.2
Smooth	0.2	0.1	0.2	0.2
Thickness Rendering	0.1	0.1	0.6	0.5
Normal Restore	0.1	0.1	0.1	0.1
Shading	0	0.1	0.1	0.1
All	40.5	43.1	44.6	44.9

续表 4.3

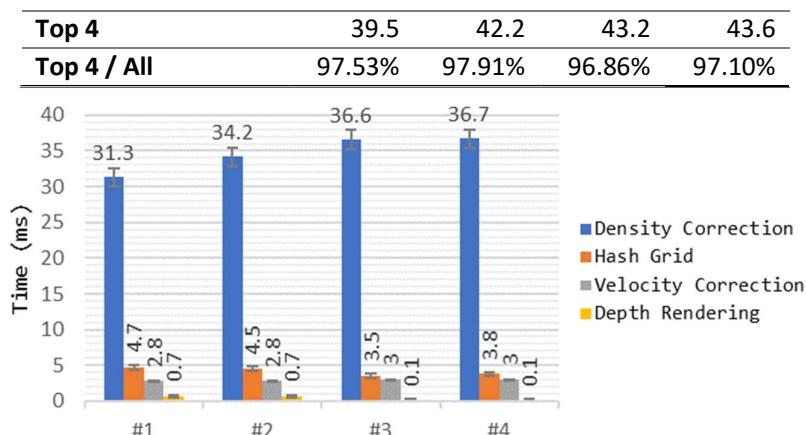


图 4.5 四个最长耗时模块在四个场景下的耗时统计

从测试结果可以看出，系统完成一帧的模拟和渲染耗时大概在 40 毫秒至 45 毫秒之间，约合 20 至 25 帧每秒，基本满足了本文开题时设定的实时性目标。同时，注意到密度修正、哈希网格建立、速度修正和深度渲染消耗了一帧中绝大部分的时间，这其中密度修正又消耗了超过 80% 的时间。

图 4.5 展示了四个模块在四个不同场景下的耗时统计。注意到，密度修正模块在恢复阶段和平稳阶段明显比下落时间和震荡阶段所花费的时间长。这是因为，液体粒子的平均距离在后两个阶段中更短，因此每个粒子有更多的邻居，每个 CUDA 并行线程需要花费的邻居遍历时间也相对变长。与之相对的，在平稳下落场景中，粒子间距等于核半径大小，粒子间几乎没有相互作用；在震荡场景中，粒子分布较为散乱，不如后两个场景集中。

后两个场景集中的粒子也意味着液体总体占据的哈希格子数量较前两个场景少。更少的哈希格子能提升 GPU 缓存命中的几率。反映在实验中，则可以发现后两个场景的哈希格子建立时间较前两个场景的更短。

装

订

线

## 5 结论和展望

### 5.1 结论

针对流体的实时模拟渲染问题，本文给出了 PBF 模拟与屏幕空间渲染算法的并行实现。该方案能够在严苛的实时应用场景下，提供真实可信的液体模拟和渲染结果。

实时性上，本文使用了 CUDA 并行框架和 OpenGL GLSL 语言作为并行计算工具，实现了流体模拟和渲染全流程的并行。另外，本文针对 GPU 架构的特点，使用了 Block 级别共享内存、循环展开、剔除分支等技术，充分挖掘了 GPU 的指令优化和存储结构优化空间。实验表明，本文给出的算法能够在 GTX 1050 GPU 上实现 32K 粒子的实时模拟。

真实性上，本文在[24]的框架上提出使用双边滤波作为液体深度纹理平滑的方法。实验表明，该方法在还原平滑的液体表面的同时，很好的保持了液体的边缘，避免了边缘处理不当造成的层次感缺失。最后，本文综合了菲涅尔方程，比尔-朗博定律和光线追踪方法，给出了一个屏幕空间液体液体表面着色模型。实验表明，该模型能够正确反映真实世界中液体表面在不同厚度，观察角度和环境光下的着色变化，给出了具有真实感的液体表面着色效果。

### 5.2 展望

#### 5.2.1 模拟部分

本文的工作基于 PBF 模拟方法。作为一种拉式方法，该算法也具有拉式算法常见的通病。首先，算法不能很好的捕捉流体的旋度特征。包括漩涡、卷起的浪花等液体现象，本文的工作表现得粗糙，甚至无法体现。可考虑在流体速度修正中加入[15]介绍的旋度修正，或可考虑通过增加粒子数量提升模拟精度。本文使用 GTX 1050 仅能实现 32K 个粒子的实时模拟。为了实时模拟更多的粒子，可使用预算算纹理，通过纹理查表，替换掉现场计算的平滑核函数以提升效率。2015 年，Wang[47]提出了一种利用切比雪夫半迭代法替代雅可比迭代以加速 PBD 的方法，并且这个方法适用于 GPU 并行。考虑到密度修正占用了模拟的大部分时间，将该算法应用在 PBF 中有相当的前景。

本文的边界形式比较简单。可以考虑实现流体与任意形状的刚体碰撞。可利用带符号的距离场（Signed Distance Field）[5]将刚体离散为体素，体素记录到最近刚体平面的距离，实现粒子与任意形状的刚体碰撞。也可以考虑实现流体与刚体的双路交互。例如，[40]提出将刚体离散成粒子，在 PBD 的框架内处理粒子间约束。

#### 5.2.2 渲染部分

本文没有使用点光、方向光等固定光源，因此没有实现阴影。在有固定光源的情形下可使用阴影纹理等方法实现阴影。[48]提出了一种光源空间焦散模拟方法，理论上可以应用在屏幕空间液体渲染当中。加入焦散能够极大地增强液体的视觉真实性。[49]提出了一种辅助粒子方法，能够根据 SPH 的液体粒子性质生成随液体流动的水花、泡沫和气泡粒子。最后，本文实现的模

订

线

拟方法不应囿于液体的模拟，目前将 PBF 方法拓展到气体和大黏性的流体的研究十分有限，因此这将是一个有价值的研究方向。

装

订

线

## 参考文献

- [1] J. Stam, “Stable fluids,” in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, 1999, pp. 121–128.
- [2] R. Fedkiw, J. Stam, and H. W. Jensen, “Visual simulation of smoke,” in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, 2001, pp. 15–22.
- [3] F. Losasso, F. Gibou, and R. Fedkiw, “Simulating water and smoke with an octree data structure,” in *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*, 2004, vol. 23, no. 3, p. 457.
- [4] B. E. Feldman, J. F. O'Brien, B. M. Klingner, and T. G. Goktekin, “Fluids in deforming meshes,” in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, 2005, p. 255.
- [5] R. Bridson, *Fluid Simulation for Computer Graphics*. A K Peters, 2007.
- [6] A. Chern, F. Knöppel, U. Pinkall, P. Schröder, and S. Weißmann, “Schrödinger's smoke,” *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–13, Jul. 2016.
- [7] R. A. Gingold and J. J. Monaghan, “Smoothed particle hydrodynamics: theory and application to non-spherical stars,” *Mon. Not. R. Astron. Soc.*, vol. 181, no. 3, pp. 375–389, Dec. 1977.
- [8] L. B. Lucy, “A numerical approach to the testing of the fission hypothesis,” *Astron. J.*, vol. 82, p. 1013, Dec. 1977.
- [9] M. Desbrun and M.-P. Gascuel, “Smoothed Particles: A new paradigm for animating highly deformable bodies,” Springer, Vienna, 1996, pp. 61–76.
- [10] M. Müller, D. Charypar, and M. Gross, “Particle-Based Fluid Simulation for Interactive Applications,” *Proc. 2003 ACM SIGGRAPH/Eurographics Symp. Comput. Animat.*, no. 5, pp. 154–159, 2003.
- [11] B. Solenthaler and R. Pajarola, “Predictive-corrective incompressible SPH,” in *ACM SIGGRAPH 2009 papers on - SIGGRAPH '09*, 2009, p. 1.
- 装  
[12] C. Huang, J. Zhu, H. Sun, and E. Wu, “Parallel-optimizing SPH fluid simulation for realistic VR environments,” *Comput. Animat. Virtual Worlds*, vol. 26, no. 1, pp. 43–54, 2015.
- 订  
[13] X. Nie, L. Chen, and T. Xiang, “Real-time incompressible fluid simulation on the GPU,” *Int. J. Comput. Games Technol.*, vol. 2015, 2015.
- 线  
[14] B. Solenthaler and M. Gross, “Two-scale particle simulation,” *ACM Trans. Graph.*, vol. 30, no. 4, p. 1, 2011.
- [15] M. Macklin and M. Müller, “Position based fluids,” *ACM Trans. Graph.*, vol. 32, no. 4, p. 1, Jul. 2013.
- [16] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, “SPH Fluids in Computer Graphics,” *Eurographics*, no. 2, pp. 21–42, 2014.
- [17] F. H. Harlow, “The particle-in-cell method for numerical solution of problems in fluid dynamics,” in *Proceedings of Symposia in Applied Mathematics*, 1963.
- [18] J. U. Brackbill and H. M. Ruppel, “FLIP: A method for adaptively zoned, particle-in-cell calculations

装  
订  
线

- of fluid flows in two dimensions,” *J. Comput. Phys.*, vol. 65, no. 2, pp. 314–343, Aug. 1986.
- [19] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, “The affine particle-in-cell method,” *ACM Trans. Graph.*, vol. 34, no. 4, p. 51:1-51:10, Jul. 2015.
- [20] C. Fu, Q. Guo, T. Gast, C. Jiang, and J. Teran, “A polynomial particle-in-cell method,” *ACM Trans. Graph.*, vol. 36, no. 6, pp. 1–12, Nov. 2017.
- [21] Stuart Wolpert, “UCLA mathematicians bring ocean to life for Disney’s ‘Moana’ | UCLA,” 2017. [Online]. Available: <http://newsroom.ucla.edu/stories/ucla-mathematicians-help-bring-the-ocean-to-life-for-disneys-hit-movie-moana>. [Accessed: 14-Mar-2018].
- [22] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*, 1987, pp. 163–169.
- [23] W. Bethel and J. Shalf, *Visualization Handbook*. 2005.
- [24] W. J. van der Laan, S. Green, and M. Sainz, “Screen space fluid rendering with curvature flow,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*, 2009, p. 91.
- [25] C. Wojtan, N. Thürey, M. Gross, and G. Turk, “Physics-inspired topology changes for thin fluid features,” *ACM Trans. Graph.*, vol. 29, no. 4, p. 1, 2010.
- [26] M. Müller, S. Schirm, and S. Duthaler, “Screen space meshes,” *ACM SIGGRAPH / Eurographics Symp. Comput. Animat.*, pp. 9–15, 2007.
- [27] J. F. Annett, “Superconductivity, Superfluids and Condensates,” *Oxford Master Ser.*, no. May, p. 140, 2004.
- [28] N. Prize and N. Prize, “Bose-Einstein Condensation in Alkali Gases 1.,” *Physics (College. Park. Md.)*, pp. 1–14, 2001.
- [29] J. J. Monaghan, “Smoothed Particle Hydrodynamics,” *Annu. Rev. Astron. Astrophys.*, vol. 30, no. 1, pp. 543–574, 1992.
- [30] J. Anderson Jr, *Fundamentals of Aerodynamics*, vol. Third Edit. 1985.
- [31] J. Bender, M. Müller, and M. Macklin, “A Survey on Position Based Dynamics, 2017,” *Tutor. Proc. Eurographics*, 2017.
- [32] 同济大学计算数学教研室, *现代数值计算 (第2版)*, Second Edi. Beijing: 人民邮电出版社, 2014.
- [33] J. J. Monaghan, “SPH without a Tensile Instability,” *J. Comput. Phys.*, vol. 159, no. 2, pp. 290–311, 2000.
- [34] S. Clavet, P. Beaudoin, and P. Poulin, “Particle-based viscoelastic fluid simulation,” *Proc. 2005 ACM ...*, p. 219, 2005.
- [35] I. Alduán and M. A. Otaduy, “SPH granular flow with friction and cohesion,” in *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '11*, 2011, p. 25.
- [36] N. Bell, Y. Yu, and P. J. Mucha, “Particle-based simulation of granular materials,” in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '05*, 2005, p. 77.

- [37] S. Green, “Particle Simulation using CUDA,” *cse.uaa.alaska.edu*, no. September, pp. 1–12, 2013.
- [38] T. Tajima, “Plasma physics via computer simulation,” *Comput. Phys. Commun.*, vol. 42, no. 1, pp. 151–152, 1986.
- [39] H. Schechter and R. Bridson, “Ghost SPH for animating water,” *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–8, Jul. 2012.
- [40] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, “Unified particle physics for real-time applications,” *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–12, Jul. 2014.
- [41] W. Nvidia, N. Generation, and C. Compute, “Fermi white paper,” *ReVision*, vol. 23, no. 6, pp. 1–22, 2009.
- [42] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, “Fermi GF100 GPU architecture,” in *IEEE Micro*, 2011, vol. 31, no. 2, pp. 50–59.
- [43] NVIDIA, “Cuda C Programming Guide,” *Program. Guid.*, no. September, pp. 1–261, 2015.
- [44] J. A. Sethian and P. Smereka, “Level Set Methods for Fluid Interfaces,” *Annu. Rev. Fluid Mech.*, vol. 35, no. 1, pp. 341–372, 2003.
- [45] J. Kessenich, D. Baldwin, and R. Rost, “The OpenGL ® Shading Language,” *Language (Baltim.)*, vol. 1, pp. 1–29, 2010.
- [46] S. H. Ahn, “OpenGL Projection Matrix,” 2018. [Online]. Available: [http://www.songho.ca/opengl/gl\\_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html).
- [47] H. Wang, “A chebyshev semi-iterative approach for accelerating projective and position-based dynamics,” *ACM Trans. Graph.*, vol. 34, no. 6, pp. 1–9, Oct. 2015.
- [48] C. Wyman and S. Davis, “Interactive Image-Space Techniques for Approximating Caustics,” in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, 2006, pp. 14–17.
- [49] M. Ihmsen, N. Akinci, G. Akinci, and M. Teschner, “Unified spray, foam and air bubbles for particle-based fluids,” *Vis. Comput.*, vol. 28, no. 6–8, pp. 669–677, Jun. 2012.
- 装  
[50] 费昀, “不可压平滑粒子流体动力学算法GPU并行加速及其应用研究,” 清华大学本科毕业论文训练, 2013.

订

线

## 谢 辞

离线渲染、管线渲染和物理模拟是计算机图形学的三块基石，本文是我对后两者的初次探索和尝试。感谢我的指导老师，赵君峤老师开设的计算机图形学课程。在这门课上我初次领略了图形学的纷繁风采。赵老师一丝不苟的治学态度给我留下了深刻的印象，并将对我今后的人生产生深刻影响。

感谢我的评阅老师，何良华老师对我的关心以及人生上的指导。感谢沈坚老师在计算机系的辛勤耕耘。没有任何老师的汇编训练和沈老师数门课的工程训练，本文上百 KB 的并行计算代码不可能调试成功。

感谢瑞士洛桑联邦理工大学的 Wenzel Jakob 老师和它的 Mitsuba 渲染器，NanoGUI 界面库和著作《基于物理的渲染》。在他的课上我深入了解了光线追踪技术的原理与实现，并有幸在物理渲染比赛中获奖。感谢他在图形学研究方向上给我的建议，促成了本文物理模拟的主题。

感谢素未谋面的哥伦比亚大学费哟。他高水平的本科毕业论文[50]成为本论文立题的契机。

感谢瑞士洛桑联邦理工大学 Wenzel Jakob, Matthias Grossglauser 老师，数据库课的关佶红老师，中科院软件所的魏峻老师，为我写研究生申请的推荐信。他们的肯定和支持给了我出国读研的信心和勇气。

感谢我的女友，在我最困难的时候不离不弃，在这条前路未卜的道路上为我提供持续前行的动力。

感谢我的父母无条件的理解和支持。他们是我最坚强的后盾。

感谢同济数理强化班和计算机系所有的朋友，与我一起并肩战斗过的伙伴，关心我的同学和老师。你们让我度过了最饱满，最激情的四年大学时光。

四年弹指一挥间，我的初心从未改变。

装

订

线