不可压流体的模拟、渲染和应用研究

摘 要

不可压流体

**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

**The simulation, rendering and application of incompressible fluids**

**ABSTRACT**

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

目 录

[1 引 言 1](#_Toc402184259)

[1.1 研究背景 1](#_Toc402184260)

[1.2 流体模拟的研究现状与相关工作 1](#_Toc402184261)

[1.3 流体渲染的研究现状与相关工作 1](#_Toc402184264)

[1.4 本文所作的工作 1](#_Toc402184265)

[2 流体的PBF模拟方法 2](#_Toc402184266)

[2.1 Navier-Stokes方程组 2](#_Toc402184267)

[2.2 平滑粒子动力学模型 2](#_Toc402184268)

[2.3 求解不可压性的PBF方法 2](#_Toc402184268)

[2.3.1 流体位置的压强修正 2](#_Toc402184269)

[2.3.2 流体速度的涡度、粘度修正 2](#_Toc402184270)

[2.4 PBF方法的GPU并行化实现 2](#_Toc402184268)

[2.4.1 CUDA并行编程简介 2](#_Toc402184269)

[2.4.2 近邻粒子寻找算法 2](#_Toc402184269)

[2.4.3 实现管线细节 2](#_Toc402184269)

[3 拉式流体的屏幕空间渲染 3](#_Toc402184272)

[4 结果和讨论 4](#_Toc402184280)

[5 结论和展望 6](#_Toc402184288)

[5.1 结论 6](#_Toc402184289)

[5.2 展望 6](#_Toc402184290)

[参考文献 7](#_Toc402184291)

[谢 辞 8](#_Toc402184292)

# 1 引 言

## 1.1 研究背景

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。与这些领域不同，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、医疗研究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一些问题[5]。2016年，Chern等人发表了论文《薛定谔的烟》[6]，提出使用为值域的波函数描述流体状态，并使用凝聚态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）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。随后Macklin等人的基于位置的流体（Position base fluids），又在同等精度下提高了模拟步长和稳定性，进一步提升了流体模拟的实时性。该方法是本文实现的重点。

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）[11]，和隐式粒子流体法（FLuid Implicit Particle, FLIP）[12]。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）[13]，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）[14]，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》）[15]。

## [1.3 流体渲染的研究现状与相关工作](#生成函数法及其优势)

## 1.4 本文所作的工作

本文针对不可压流体，实现了一种新的拉式（Lagrangian）实时模拟方法，称为基于位置的流体（Position based fluids）方法[16]。同时，本文实现了一种屏幕空间的拉式流体渲染方法，这种方法利用深度缓存（Depth buffer）还原粒子化表示的液体[17]，省去了传统方法中的重建步骤，表现出优秀的实时效果。

# 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(a), (b)，体积V，dV）

首先我们考虑空间中的一块流体区域，这块区域内的任意一点有密度、速度等物理量。设外力为，对这一块流体区域应用**牛顿第二定律**，有

注意（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，区域是随时间变化的量。（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，是液体的任意一个物理量，是区域的边界，有

其中方程右侧 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。方程左侧物理量在整块区域的变化，是右侧第一项区域内随时间变化，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（1）左侧，可以得到

上式比较复杂，但实际上，我们可以利用质量守恒定律做简化。空间虽然会随时间发生变动，但我们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**

对质量守恒定律应用莱布尼茨法则，有

因为对任意的上面的等式都成立，必有，而这一项恰好出现在（3）中，因此可将（3）简化。将简化的（3）带回（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**。

如果将（5）与牛顿第二定律方程的基本形式相比较，能够发现两者之间形式上的相似性：对应，同时中的正好是加速度的定义，而多出来的一项是“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。中表征了速度变化的方向，而则表征了漂移的方向。

为了简化表示，定义**材料微分**运算符

利用（6）将（5）重写为（7）

现在考虑作用在体积上的外力，它们的存在使得液体的速度发生改变。首先容易想到，应包含有重力，其中是重力常数。除此之外，流体会还受到周围流体的挤压，产生压强力。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。下面我们将逐项分析。

首先考虑压强力。压强仅在体积的表面对有贡献，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1.(c)，片元压强）

对于表面上的任意一块小片元，设此处压强为，表面法向量为，由压强的定义容易知道。因此作用在整块体积上的压强力为

上式中的第二个等号应用了散度定理。

最后我们来考虑粘性力。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[18]。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[19]。

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。如下图所示，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。

（图2-2. 断层上的粘性力）

流体速度的梯度表征空间不同方向上流速变化的强度和方向。在区域边界上任意一处片元，我们考虑粘性力对体积的作用。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方向变化，此反作用力没有离开表面，因此此力与其反作用力对体积没有影响。由此推理，对体积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。应当注意到，在三维空间中，是一个3x3张量，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。

在表面上对粘性力积分就得到了作用在整个体积上的粘性力，由如下公式给出

其中是**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，因此我们假设一种流体的粘度始终是一个常数。

综合重力，压强力和粘性力，带入动量方程（5）中，我们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。我们将在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。

**2.1.2 不可压条件**

在密度，重力常数，粘度都给定的情况下，动量方程（11）仍并不足以给出流体的一个解，因为流体的压强是一个随时间和位置变化的变量。向量方程（11）在x, y, z上给出三个标量方程，但速度和压强构成了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。

动量方程（11）和不可压条件（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，对一块我们确定的体积，其中含有固定的流体分子，无论体积的形状怎么变化，它的**大小**都不会发生改变。考虑一段时间内体积的变化量与其表面处液体流速的关系

第一个等号是因为质量守恒=0，第二个等号参看图1(b)，第三个等号应用了散度定理。因为对任意上式都成立，所以有不可压条件（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。（TODO：关于点云、溅射、图）

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[20]，但通常支撑集是有限的，并将支撑集的半径记为。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。
2. 高阶矩等于零， 。
3. 支撑集有限，对于某个有。  
   这个条件使得计算空间某点的物理量时仅用考虑距离在范围内的粒子。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，**。**
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。

(图2-3，典型的核函数)

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算法的密度估计和梯度计算。Poly6核的定义如下

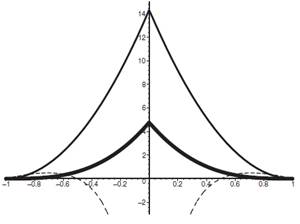
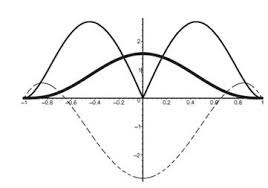
Poly6核的梯度为

Poly6核的梯度仅含有的二次项，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），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，我们使用Spiky核用于计算核函数的梯度。Spiky核的定义如下

Spiky核的梯度为

注意到Spiky函数是一个向量函数，且在处函数值的长度达到最大。



（图2-4，h=1 ）

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可通过对整个空间中粒子的核函数求和近似得到。

通过定义（16）可以注意到，一个粒子中心的物理量并不等于这个粒子携带的物理量。因为其他位置的粒子也会对这个位置有所贡献。

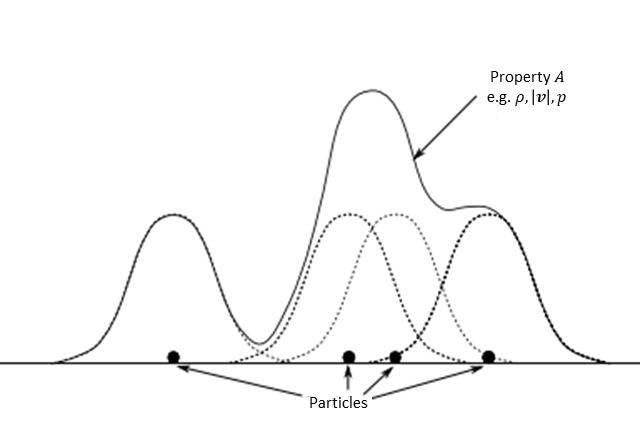


图2.5 空间离散粒子还原空间连续物理量

因为模拟的物理量离散地记录在粒子上，我们通常不关心任意位置的物理量，而只关心粒子中心处地物理量。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。

粒子位置的密度为

其中。SPH模拟常常还需要计算物理量的梯度。式（17）为书写方便定义在上，但所有其他粒子的位置都是密度的隐含自变量。因此式（17）的梯度分为关于的梯度和关于的梯度。

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，其中是一个跟气体比热容有关的绝热因子，是每单位质量的气体内能[21]。对于液体，计算机图形学通常使用可压性更弱的Tait方程。Tait方程有如下的形式

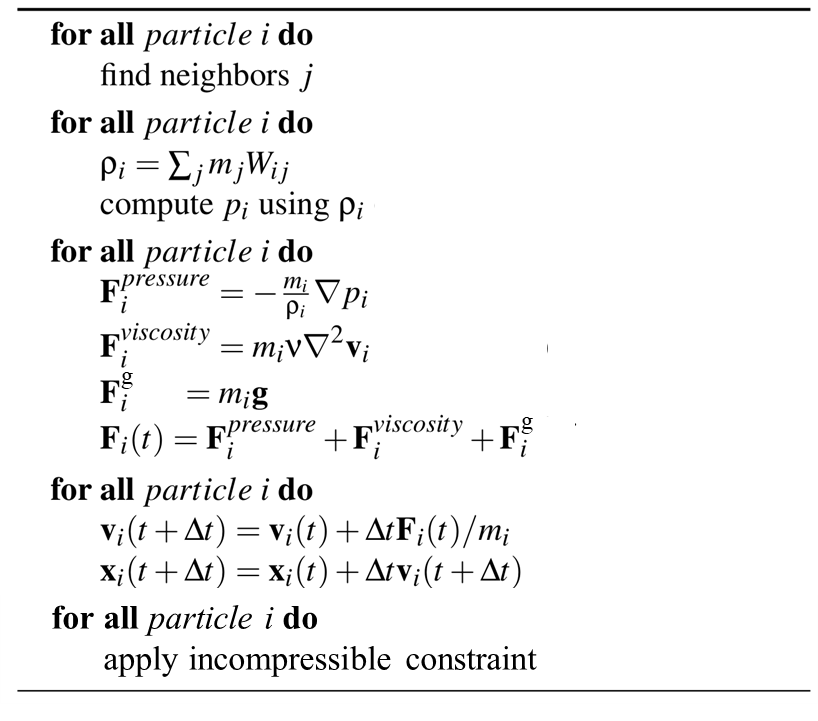
密度导出的压强是许多SPH方法中计算压强力的核心。但本文实现中使用的方法没有直接应用到密度导出的压强，因此不再做展开。

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，考虑对时间的导数，其中是粒子位置，注意到也是时间的函数，有

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，我们给出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，粘性力和重力，直接利用动量方程更新速度，最后根据不可压条件修正粒子速度和位置。



算法1.

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，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，本文的重点之一，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[16]。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（Preditctive-correcive SPH，PCISPH）等传统的SPH方法则是形式化地引入压强力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[22]。BPF方法避免了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Jacobi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[23]。

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（18）中我们提到，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。对粒子定义为EOS约束

其中是液体的静止密度。当液体平衡时，必有。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（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（19）求解，经过有限次数的迭代后必然能满足。

为了通过EOS方程（19）更新位置，首先注意到是粒子位置的函数，因此也是粒子位置的函数，设粒子个数为，有

其中是邻居算子，表示在以位置为中心，核半径为大小的区域内的粒子的集合。为计算方便我们假设粒子质量都为1千克，即。我们期望经过一个粒子位置修正后，每个粒子的EOS约束能被满足，即

可应用梯度下降法，让与在处的梯度平行，设学习率。一种朴素的方法是，设定为某个预设值，迭代更新粒子位置，直到方程(22)达到某个精度阈值。这种方法在实际中并不可行，因为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。

实际上可以利用方程（22）的一次微分展开估计**。**注意到方程(22)可微，有一次近似

以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，若可逆，则方程有确定解。但是由于的维度是，在实际中通常大于，而最好的的矩阵求逆算法复杂度高达，完全无法承受，因此也不能通过直接求解估计**。**

作为方程（23）的近似解法，我们考虑高阶线性方程组的Jacobi雅可比迭代法[24]。Jacobi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Jacobi迭代法的更新公式为

同时这里给出的公式。

根据和的相同和不同，又有两种形式

公式（24）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（24）做修正。我们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。在此我们参照[16]中的办法，直接在分母处引入一个（相对梯度和）小的常数。在我们的最终实现中取。这个值在数值上并不小，但相对于Spiky梯度和来说是小的。修正后的计算公式为

公式（25）对粒子仅根据和更新其位置。换句话说，仅根据粒子本身的信息，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（25）略作修改，不仅考虑粒子本身，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

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。我们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，我们在实现中设定的其他常数参数包括核半径，静止密度，重力，模拟时间步长（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。

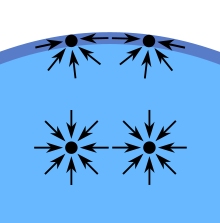
 

图. 表面张力形成原理

仿照表面张力的形成原理，可以在SPH中通过通过引入分子间作用力实现表面张力的效果，论文[25]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[26][27][28]。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，[16]中引入了最早在[25]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。对粒子和粒子，令虚拟压强系数为

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（29）变为

在实现中我们取。

2.3.3 流体的粘度修正

为了还原动量方程中的粘性力，我们引入对粒子速度引入一个由相对速度引起的粘度修正。粘度修正是Navier-Stokes方程中粘性力的近似。

公式（32）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，微元表面积与密度和满足（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（32）在分母位置引入的密度和是对[16]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。

在实现中我们取。

2.3.4 流体的涡度修正

TODO

## 2.4 PBF方法的GPU并行化实现

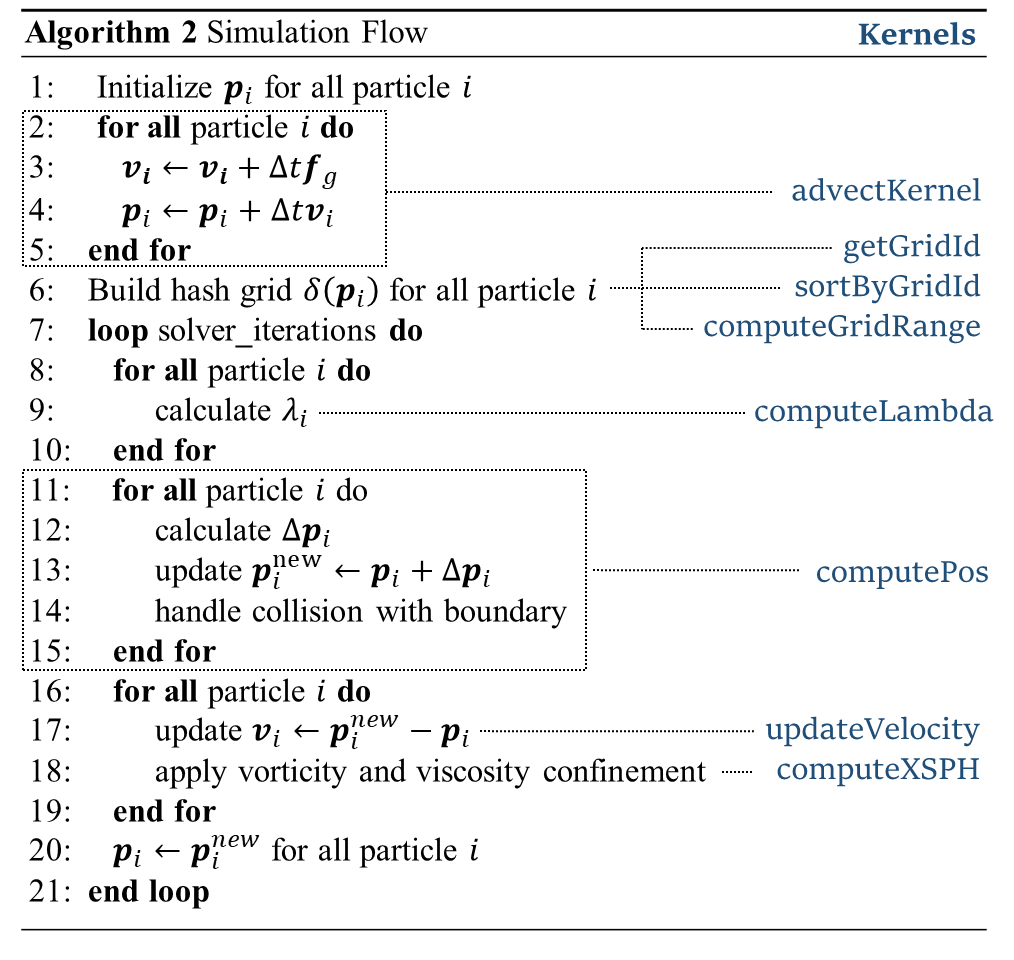
本节叙述 PBF 方法的 GPU 并行化算法的流程、设计与实现。我们首先概述 PBF 方法的算法流程，指出 PBF 高度适合并行化计算的特征，并对邻居查找、迭代方式、边界处理等几个关键设计做简要说明。2.4.2节介绍通用GPU 计算（General Purpose GPU, GPGPU）和本文使用的Nvidia CUDA 并行语言。2.4.3节介绍CUDA核函数和数据结构设计。2.4.4节介绍本文使用的哈希网格邻居查找算法，2.4.5节对算法管线的实现细节做进一步说明。

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[29]。之后，根据公式（31）迭代地使用Jacobi 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和涡度修正。

算法2. PBF模拟算法流程



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**：我们首先使用欧拉积分，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）[30]。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 方法的粒子核半径是有限的，两个中心距离大于的粒子对互相没有作用，因此产生了粒子邻居的概念。记是中心位置在的粒子的邻居几何，任意邻居到的距离不超过。在利用公式（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量级，一个核直径内的粒子数量通常小于量级，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，我们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。反应在结果中产生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。

**Jacobi方法：**在2.3.1节中我们已经介绍过Jacobi 方法，并给出了使用Jacobi方法修正粒子位置的公式。Jacobi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法和超松驰(Successive over-relaxation， SOR)法等[24]。Jacobi方法相比其他迭代法的优势在于它非常适合并行化。Jacobi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，若能讲针对每个粒子，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。（Future works: A chebyshev semi-iterative approach for accelerating projective and position-based dynamics）

**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[31]补偿边界密度。最初我们的实现在刚体边界上有粒子抖动的问题，因此引入了[31]中的方法。在我们加入了粘性力后，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，因此我们最终仅引入了刚体边界碰撞这一种边界条件。

我们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。实现中我们取。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。下一节我们会提到，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。此时我们把越界的粒子归在投影后的网格内，但不做位置上的更新。

我们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[32]。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的本职工作——3D图形渲染中，为了保证画面的流畅，GPU通常需要在10毫秒级别的时间内对上万个顶点坐标做线性变换，对百万个的面元（Fragment）计算着色。因此GPU被设计成了与CPU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[33]。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。

|  |  |
| --- | --- |
|  |  |

图. Fermi GPU 架构

不同架构的Nvidia GPU 具有不同的 SM 数目，每个 SM 内的 CUDA 核心数量也不尽相同。我们实验中使用的GPU 是移动版的GTX 1050，拥有5个SM，每个SM内128个CUDA核心 ，共640个 CUDA 核心，峰值单精度浮点运算吞吐量为1733 GFLOPS[34]。

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提供了并发任务流的概念，允许不同的核函数并发执行，但我们的模拟应用仅需要使用一条任务流。同一时刻，GPU仅执行一个核函数。一个GPU核心内运行该核函数的一个线程。

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。在核函数内，可通过blockIdx.x（blockIdx.y, blockIdx.z）和threadIdx.x（threadIdx.y，threadIdx.z）获得当前线程的索引。Block和Thread的维度大小由程序员在核函数调用时指定。

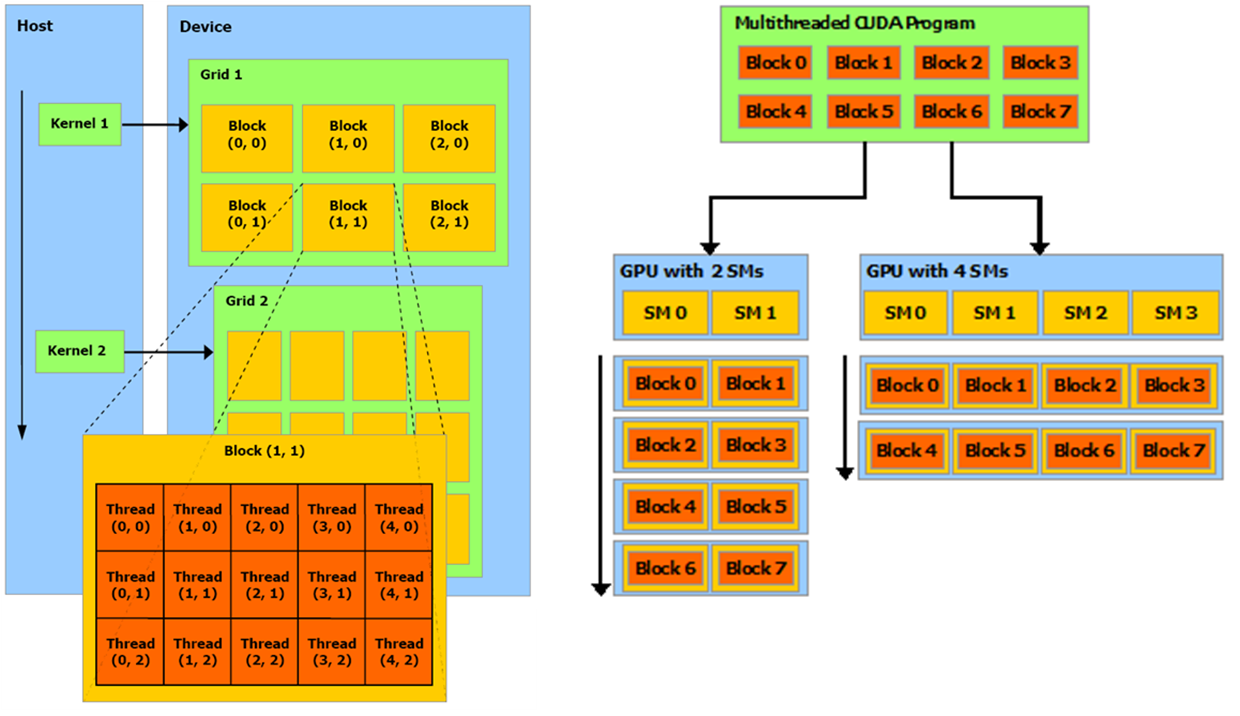


图. CUDA核函数组织和执行层次。左：并行组织；右：执行分配

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现存上，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，我们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. PBF算法用数据结构

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 位置 | 描述 |
| dc\_pos | float3 \* | OpenGL | 粒子位置 |
| dc\_npos | float3 \* | OpenGL | 备用粒子位置 |
| dc\_tpos | float3 \* | CUDA | 临时粒子位置 |
| dc\_vel | float3 \* | CUDA | 粒子速度 |
| dc\_nvel | float3 \* | CUDA | 备用粒子速度 |
| dc\_iid | uint \* | OpenGL | 粒子编号 |
| dc\_gridId | uint \* | CUDA | 粒子所属网格编号 |
| dc\_gridStart | uint \* | CUDA | 网格起始索引\* |
| dc\_gridEnd | uint \* | CUDA | 网格终止索引\* |
| dc\_lambda | float \* | CUDA | 粒子所在处值 |
| dc\_pho | float \* | CUDA | 粒子所在处值 |

\*  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. PBF算法用CUDA核函数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模块 | 名称 | 输入 | 输出 | 描述 |
| advect | advect\_kernel | dc\_pos, dc\_vel | dc\_npos | Leapfrog积分 |
| buildGridHash | thrust::  transform | d\_npos | d\_gridId | 计算粒子所属网格编号 |
| 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 | 计算网格起止索引\* |
| correctDensity | computeLambda | dc\_gridId, dc\_gridStart, dc\_gridEnd, dc\_npos | dc\_lambda, dc\_pho | 计算粒子 |
| computetpos | dc\_lambda, dc\_gridId, dc\_gridStart, dc\_gridEnd, dc\_npos | dc\_tpos | 计算粒子 |
| thrust::  copy\_n | d\_tpos | d\_npos | 更新 |
| updateVelocity | thrust::  transform | d\_pos, d\_npos | d\_vel | 计算 |
| correctVelocity | computeXSPH | dc\_pho, dc\_gridStart, dc\_gridEnd, m\_gridHashDim, dc\_npos, dc\_vel, | dc\_nvel | 修正 |

2.4.4 哈希网格邻居寻找算法

在2.4.1节我们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。我们使用的是哈希网格（Hash grid）邻居查找算法[29]。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。

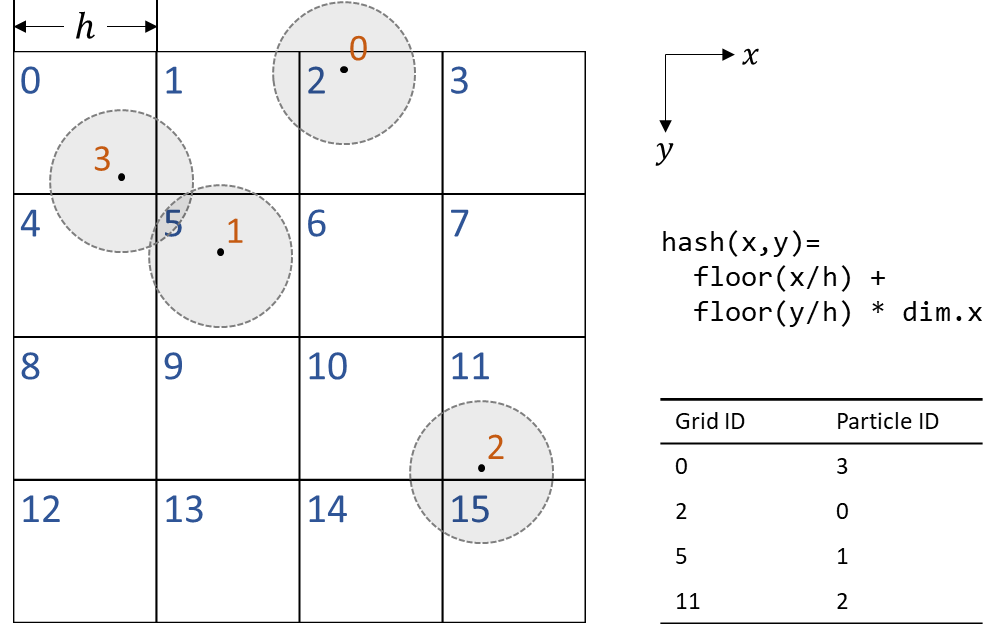


图. 哈希网格结构

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，也是我们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。但在此我们不做探究。

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。

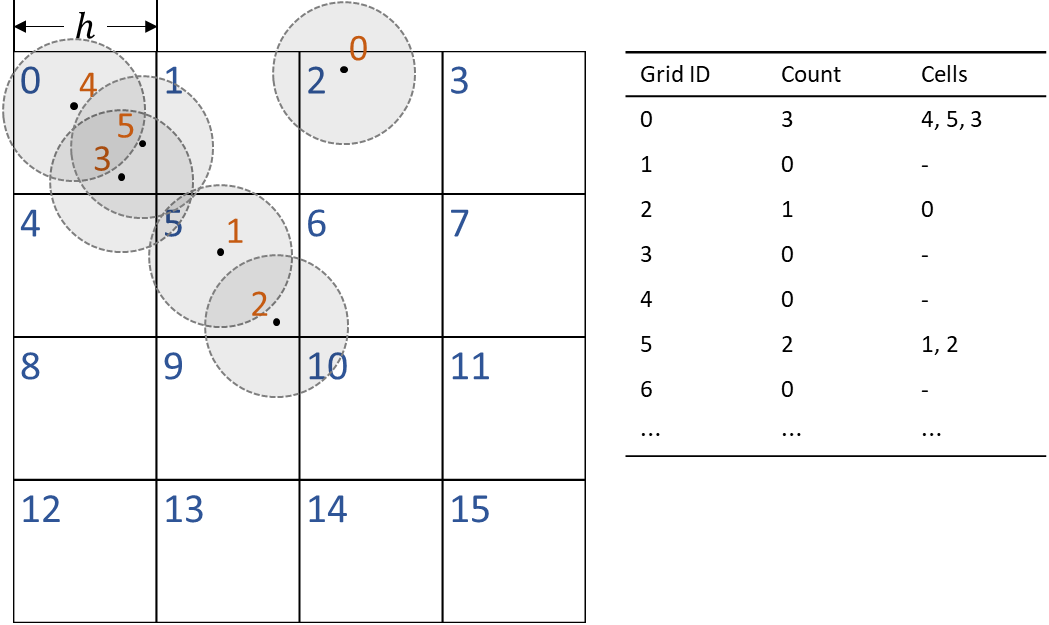


图. 原子更新格子记录

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编号为id的格子。排序使用了thrust库的GPU基数排序算法。

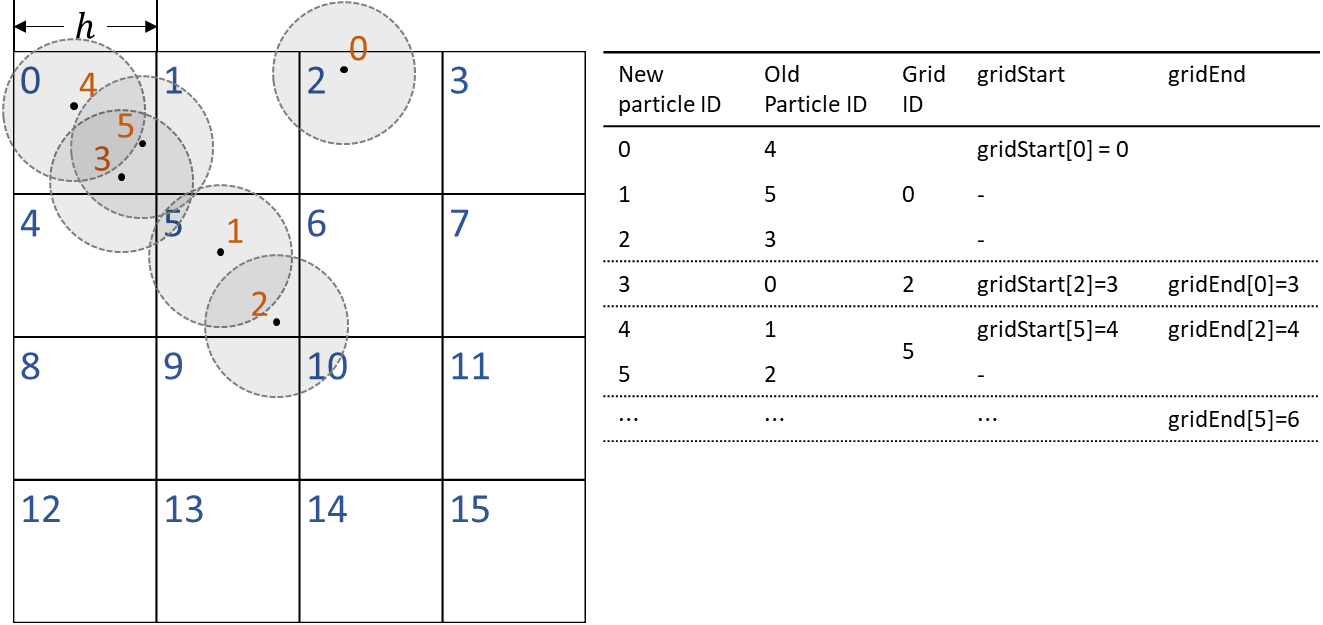


图. 排序后计算格子记录

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

**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);  }  \_\_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. 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）。这也是我们最初的做法。观察Poly6的定义可以发现，当确定时，**，**其中是某个只与有关的常数。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。我们通过实验发现这个优化能带来超过50%的帧数提升。

**双缓冲：**我们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 到 的拷贝，节省了一定的时间。渲染模拟结果时，我们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节中我们提到，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。

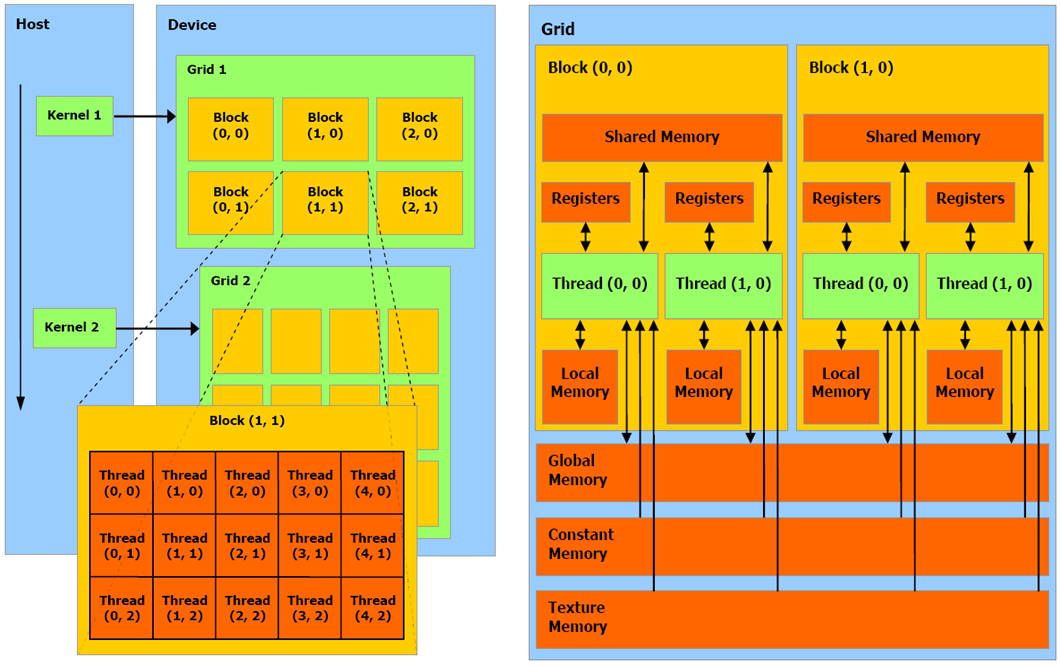
核函数寻找粒子邻居需要遍历粒子所在格子的6个邻居格子。邻居格子可能会超出立方体边界。这些越界邻居格子是没有粒子的，gridStart和gridEnd中也没有它们的记录，因此需要跳过。一种方法是，在每个维度上判断越界，尽早跳过。另一种方法则是在最内侧循环判断越界，尽量迟跳过。这两种方法在效果上是等价的，在性能上有差异。后者因为尽量减少了核函数分支分歧，能够获得更好的性能。

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;  /\* ... \*/  } |

同时我们还使用了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。下图展示了CUDA核心的内存结构。



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，我们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）。我们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。

# 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[35][17]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[36]；移动立方体（Marching cubes）法从体素中还原多边形网格（Polygonal mesh）[37]。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[17]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。算法流程如下图所示：

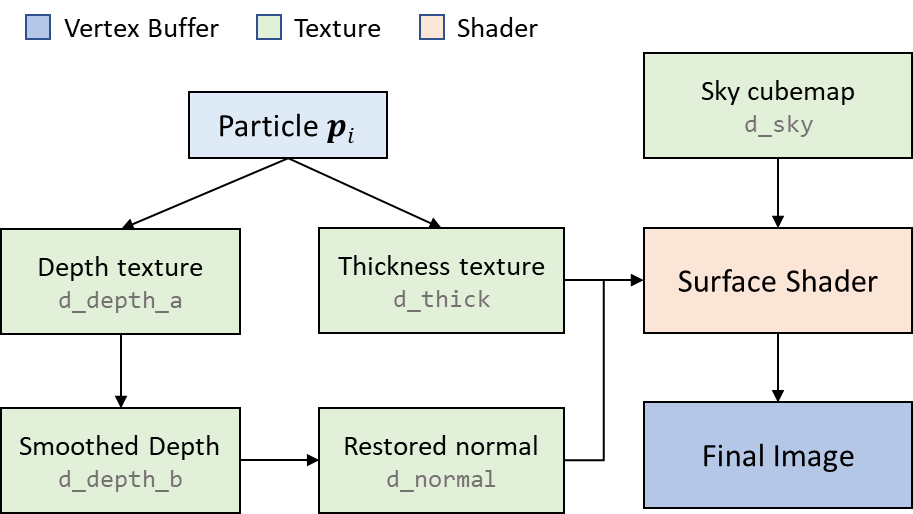


图. 渲染算法流程

渲染算法全部使用OpenGL 4.6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。但是仅就我们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。因此在我们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，我们需要使用GL\_R32F类型存储32位精度的纹理，使用GL\_RGB32F存储向量纹理。这不妨碍我们将其视作普通的float和float3类型使用。

表. 渲染算法使用的纹理

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 维度 | 类型 | 描述 |
| d\_depth | GL\_TEXTURE\_2D | GL\_DEPTH\_COMPONENT32F | 用于深度监测的深度纹理\* |
| d\_depth\_a | GL\_TEXTURE\_2D | GL\_R32F | 摄像机空间深度纹理\* |
| d\_depth\_b | GL\_TEXTURE\_2D | GL\_R32F | 备用摄像机空间深度纹理\*\* |
| d\_normal | GL\_TEXTURE\_2D | GL\_RGB32F | 法向量纹理 |
| d\_thick | GL\_TEXTURE\_2D | GL\_R32F | 厚度纹理 |
| d\_sky | GL\_TEXTURE\_CUBE\_MAP | GL\_RGB | 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。

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. 渲染算法使用的着色器

|  |  |  |  |
| --- | --- | --- | --- |
| 着色器名称 | 输入 | 输出 | 描述 |
| get\_depth | vertex\_vao | d\_depth, d\_depth\_a | 用于深度监测的深度纹理\* |
| get\_thick | vertex\_vao | d\_thick | 摄像机空间深度纹理\* |
| smooth\_depth | d\_depth\_a\* | d\_depth\_b\* | 备用摄像机空间深度纹理\*\* |
| restore\_normal | d\_depth\_b\* | d\_normal | 法向量纹理 |
| 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

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，与屏幕坐标系是不用的概念。

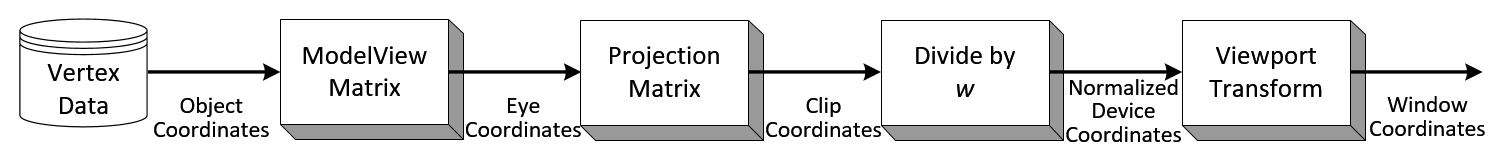


图. OpenGL顶点坐标变换过程

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，我们将顶点坐标左乘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，我们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坐标的信息。我们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我们研究的重点。

我们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轴坐标。注意在OpenGL使用的右手坐标系中，整个视体都位于轴的负半轴区域内，因此前截面的坐标为，后截面的坐标为。

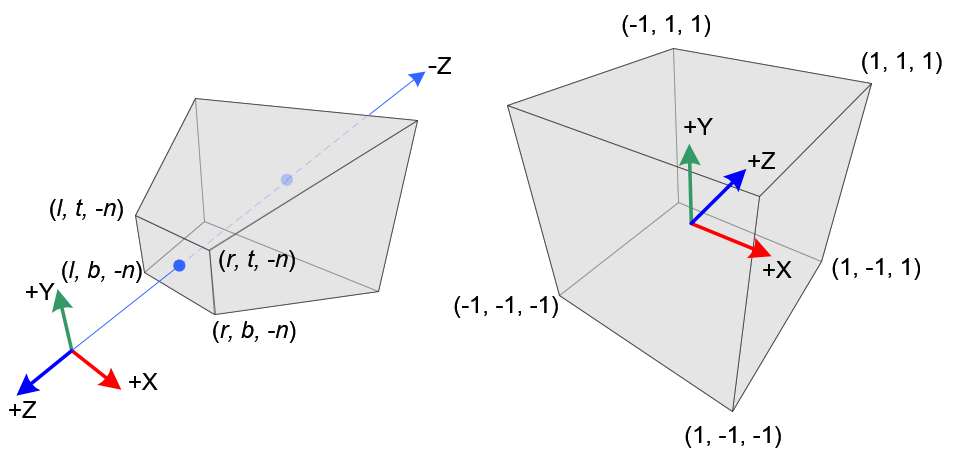


图. 投影变换的平截头体

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的正方形。下图以坐标为例，展示了视体中一点投影到前截面的过程。

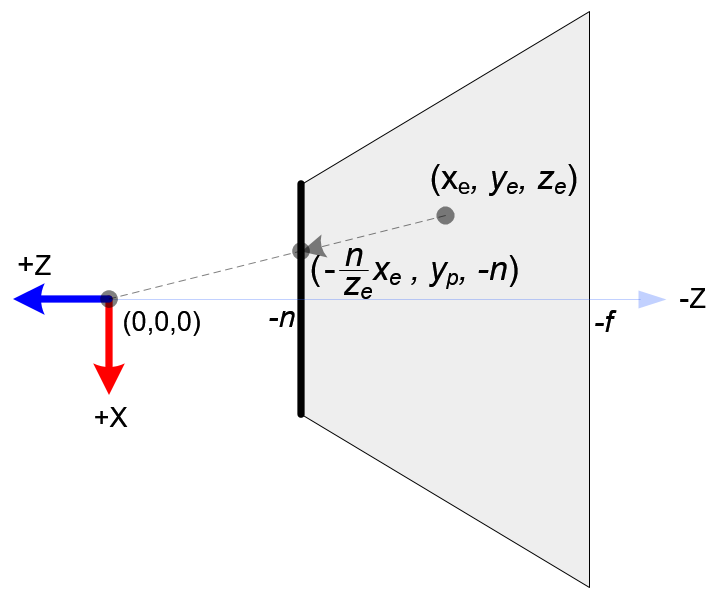


图. 透视变换坐标投影示意图

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，投影到前截面后坐标为，由相似关系容易得到

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，有



图. 视体前截面转换到NDC坐标系、贴图坐标系

通常我们希望视体中心对称，此时有。在此条件下，综合（33）到（36），我们给出从NDC坐标系还原摄像机坐标系的公式

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

在后面几节中，我们将利用公式（37）到公式（40）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，我们对液体表面的平滑、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。我们知道，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，我们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。

我们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。

下面我们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，我们使用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。为了将粒子绘制成圆形，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。为了向深度深度纹理写入深度信息，我们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。

由于我们独立地绘制顶点，而不是绘制多边形，透视投影的效果无法体现。因此我们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，用类似公式（37）的推倒过程，我们可以得到正确的quad大小

其中的定义同公式（37），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]。我们可以利用gl\_PointCoord将圆形外的部分剔除，如下图所示。

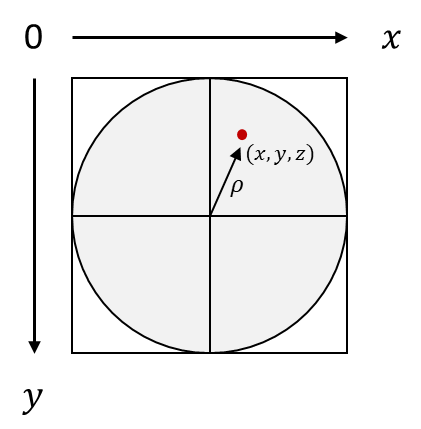


图. 在quad中还原球面坐标

舍弃的像素即可将粒子还原成圆形。同时已知粒子中心深度，粒子大小和像素坐标，由初等立体几何可以得到该像素点的深度。将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会先判断当前像素和深度测试缓存中的深度大小[[1]](#footnote-1)[38]。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。开启深度测试使得我们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。这样复杂的光线路径即使是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，我们认为屏幕空间渲染是可以胜任的。

下面我们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，我们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。设球面上坐标为的像素厚度为，球的半径为，有  
 。

A picture containing sky, outdoor, snow, nature

Description generated with very high confidence A picture containing sky, outdoor

Description generated with very high confidence

图. 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]

3.4 表面法向量重建

本节我们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，若这个位置上有液体，设它在摄像机坐标系中的坐标为，则可以从深度纹理中读取，和则可以利用公式（39）和（40）得到。由微分几何知识，此处液体表面的法向量可从以下公式得出

可以用偏微分的一阶差分近似求公式（42）

其中，和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（42）,（43），（44），（39）和（40），可以将公式（44）进一步改写成以像素坐标表示的形式。

需要注意到的是，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取到。同时，我们将纹理的采样模式设为线性插值（Linear Interpolation, OpenGL中GL\_LINEAR标志位），这可以为我们提供微小的深度平滑效果。

由于公式（42）中在粒子的边缘处不连续，导致不可微，此公式会失效。下图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，造成下图左的边缘效果。作为改进，在边缘处使用连续一侧的偏导数。体现在公式上，则是取前后差分中较小的一个。

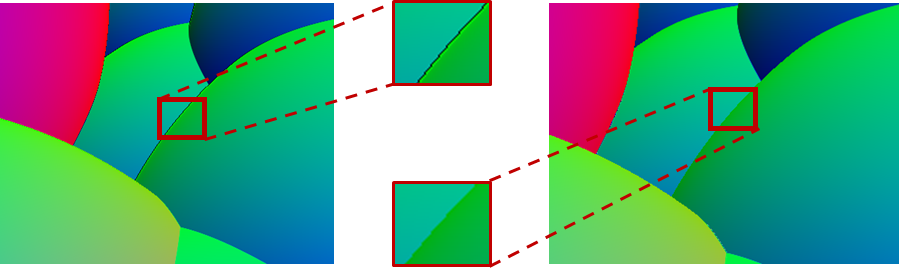


图. 边缘处的深度差分。左：未保持边缘；右：保持边缘。

A picture containing sky, outdoor, kite

Description generated with high confidence

图. 从深度纹理重建表面法向量

3.5 深度平滑

3.6 表面着色

# 3 实验部分

## 3.1 仪器和试剂

### 3.1.1 仪器

798-MPT全自动电位滴定仪（瑞士万通）；

氢离子选择性复合电极（瑞士万通）；

……。

### 3.1.2 试剂

氢氧化钠（A.R.）；

邻苯二甲酸氢钾（容量基准试剂）；

氯化钾（A.R.）；

……。

## 3.2 溶液的配制及标定

表3.1 氢氧化钠溶液厚度标定

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 邻苯二甲酸氢钾质量/g | NaOH溶液体积/mL | NaOH溶液厚度/（mol·L-1） |
| 1 | 0.1118 | 5.91 | 0.0926 |
| 2 | 0.1028 | 5.42 | 0.0929 |
| 3 | 0.1124 | 5.91 | 0.0932 |

### 

### 3.2.2 氯化钾离子强度调节剂的配制

……。

## 3.3 实验步骤

……。

# 4 结果和讨论

## 4.1 多元酸体系的结果和讨论

### 4.1.1 直接计算法

……。

### 4.1.2 半整数法

A. 半整数法的求解过程

a. 可以直接得到半整数的情况



从表4.1可见，当为0.5和1.5时的pH分别是5.32和2.74，即……。



b. 无法直接得到半整数的情况



在用半整数生成函数法直接求解时，会遇到这样的问题……。表4.4列出了滴定过程中乙二酸溶液的各主要物理量的部分数据。从表4.4可见，……。

……

……。

表4.4 草酸的部分数据列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *V*/mL | *E*/mV | pH | [H+]/（mol·L-1） |  |
| …… | …… | …… | …… | …… |
|  |  |  |  |  |
| 1.00 | 2.62×102 | 2.22 | 6.07×10-3 | 1.14 |
| 1.50 | 2.60×102 | 2.27 | 5.42×10-3 | 1.10 |
|  |  |  |  |  |
| …… | …… | …… | …… | …… |

续表4.4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *V*/mL | *E*/mV | pH | [H+]/（mol·L-1） |  |
| …… | …… | …… | …… | …… |
| …… | …… | …… | …… | …… |
| …… | …… | …… | …… | …… |

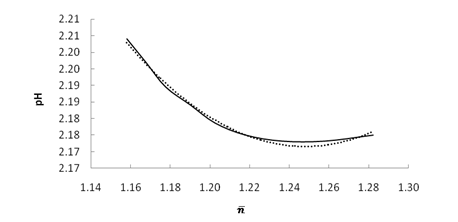


图4.1 乙二酸 ≥1.15数据段曲线及其拟合曲线（实线--实际曲线，虚线--拟合曲线）



针对这种情况，可以采用多项式拟合的方法求解。以乙二酸为例，在Excel中，选取1.28<<1.15之间的数据，以为横坐标，pH为纵坐标，做-pH曲线（见图4.1），并添加趋势线，选择相关系数*R*2最接近1的多项式作为拟合方程，……。



B. 半整数法的计算结果

利用半整数生成函数法，对各种多元酸三次平行实验数据分别进行处理，并求平均值，结果见……。

C. 半整数法计算结果的讨论

……。

### 4.1.3 分段拟合法

……。

## 4.2 氨基酸合铜体系的结果和讨论

……。

## 4.3 关于计算方法的讨论

## 4.4 关于其他问题的讨论

# 5 结论和展望

## 5.1 结论

（1）生成函数法可以分为直接计算生成函数法、分段拟合生成函数法及半整数生成函数法。这三种方法有如下特点：①……；②……；③……。

（2）本文运用三种不同生成函数法，测定了多元酸和氨基酸合铜配合物的稳定常数，得到了……。

（3）三种生成函数法中无论哪一种方法，对待测酸或配合物稳定常数的大小均有一定的要求，如……。

……。

## 5.2 展望

（1）生成函数法理论可靠，计算方便，但……。

（2）在生成函数法的应用中，还有以下问题有待研究和解决：①……；②……。

……。

# 插图索引

图2.5

图. Fermi架构[33]

图. CUDA核函数层次结构[39]

图. OpenGL顶点坐标变换过程。<http://www.songho.ca/opengl/gl_transform.html>

图. 投影变换的平截头体。<http://www.songho.ca/opengl/gl_projectionmatrix.html>

图. 透视变换坐标投影示意图。

图. 视体前截面转换到NDC坐标系、贴图坐标系

图. 在quad中还原球面坐标

图. 深度纹理和厚度纹理

图. 边缘处的深度差分

# 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] F. H. Harlow, “The particle-in-cell method for numerical solution of problems in fluid dynamics,” in *Proceedings of Symposia in Applied Mathematics*, 1963.

[12] J. U. Brackbill and H. M. Ruppel, “FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions,” *J. Comput. Phys.*, vol. 65, no. 2, pp. 314–343, Aug. 1986.

[13] 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.

[14] 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.

[15] 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].

[16] M. Macklin and M. Müller, “Position based fluids,” *ACM Trans. Graph.*, vol. 32, no. 4, p. 1, Jul. 2013.

[17] 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.

[18] J. F. Annett, “Superconductivity, Superfluids and Condensates,” *Oxford Master Ser.*, no. May, p. 140, 2004.

[19] N. Prize and N. Prize, “Bose-Einstein Condensation in Alkali Gases 1.,” *Physics (College. Park. Md).*, pp. 1–14, 2001.

[20] J. J. Monaghan, “Smoothed Particle Hydrodynamics,” *Annu. Rev. Astron. Astrophys.*, vol. 30, no. 1, pp. 543–574, 1992.

[21] J. Anderson Jr, *Fundamentals of Aerodynamics*, vol. Third Edit. 1985.

[22] J. Bender, M. Müller, and M. Macklin, “A Survey on Position Based Dynamics, 2017,” *Tutor. Proc. Eurographics*, 2017.

[23] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner, “SPH Fluids in Computer Graphics,” *Eurographics*, no. 2, pp. 21–42, 2014.

[24] 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.

[25] J. J. Monaghan, “SPH without a Tensile Instability,” *J. Comput. Phys.*, vol. 159, no. 2, pp. 290–311, 2000.

[26] S. Clavet, P. Beaudoin, and P. Poulin, “Particle-based viscoelastic fluid simulation,” *Proc. 2005 ACM …*, p. 219, 2005.

[27] 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.

[28] 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.

[29] S. Green, “Particle Simulation using CUDA,” *cse.uaa.alaska.edu*, no. September, pp. 1–12, 2013.

[30] T. Tajima, “Plasma physics via computer simulation,” *Comput. Phys. Commun.*, vol. 42, no. 1, pp. 151–152, 1986.

[31] H. Schechter and R. Bridson, “Ghost SPH for animating water,” *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–8, Jul. 2012.

[32] 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.

[33] W. Nvidia, N. Generation, and C. Compute, “Fermi white paper,” *ReVision*, vol. 23, no. 6, pp. 1–22, 2009.

[34] Nvidia, “GeForce GTX 10-Series Notebooks,” *Nvidia*, 2017. [Online]. Available: http://www.geforce.com/hardware/10series/notebook.

[35] M. Müller, S. Schirm, and S. Duthaler, “Screen space meshes,” *ACM SIGGRAPH / Eurographics Symp. Comput. Animat.*, pp. 9–15, 2007.

[36] J. A. Sethian and P. Smereka, “Level Set Methods for Fluid Interfaces,” *Annu. Rev. Fluid Mech.*, vol. 35, no. 1, pp. 341–372, 2003.

[37] 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.

[38] J. Kessenich, D. Baldwin, and R. Rost, “The OpenGL ® Shading Language,” *Language (Baltim).*, vol. 1, pp. 1–29, 2010.

[39] NVIDIA, “Cuda C Programming Guide,” *Program. Guid.*, no. September, pp. 1–261, 2015.

# 谢 辞

正文内容

1. 深度测试也有可能紧接着顶点着色之后执行（Early Fragment Test）。但由于我们改写了片元着色器的FragDepth，这种情况不会发生。 [↑](#footnote-ref-1)