

结题报告——使用MPM方法模拟雪的现象

项目地址 [lintonfirst/snow-simulation-code \(github.com\)](https://lintonfirst/snow-simulation-code (github.com))

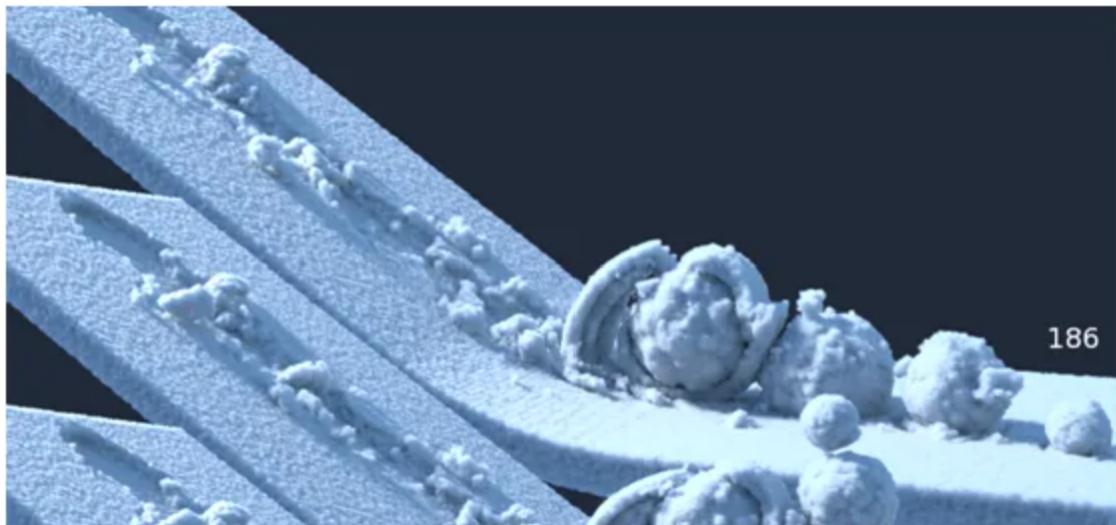
背景及研究意义

真实地模拟雪这种自然现象，一直以来都是虚拟现实和计算机图形学领域的研究热点以及难点。真实地模拟雪能够有效地增强虚拟环境的深度感和沉浸感，被广泛地应用于三维游戏、影视动画、虚拟现实等领域。

雪是一种美丽又多变的自然景观，无论是滑雪者身后飘扬的粉雪，还是踩碎冰雪外壳的脚步声，甚至是将雪卷成球堆成雪人，雪的丰富特性使得其在自然场景中有极其吸引人的视觉效果；但在计算机上建模却极其困难，很难用常规的视觉渲染方法表现其特性。



刚体摧毁城堡

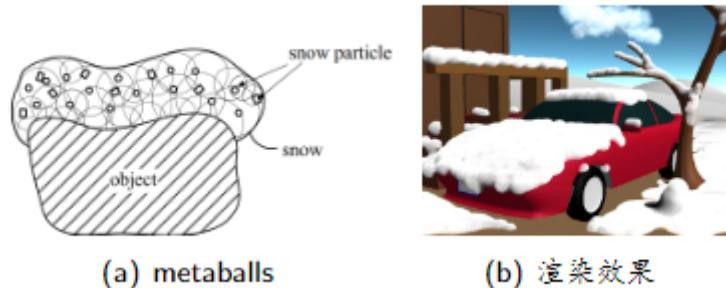


雪球从斜坡滚下

雪之所以难模拟是因为其同时具有固体和流体的特性。比如一个雪球从斜坡上滚下，雪球在被挤压结实后具有刚体的特性；但是雪球边缘处还比较松散，它与雪面之间的接触更像是一种流体运动。对此，物理仿真是需要考虑这两种特性的耦合，单用流体求解器求解无法取得理想的效果。

相关研究

A modeling and rendering method for snow by using metaballs[1997]



- 目标：模拟雪在物体表面堆积的效果
- 使用metaball来隐式建模雪的表面形状
- 使用metaball particle来处理光在雪表面发生的折射

Computer modelling of fallen snow[2000]



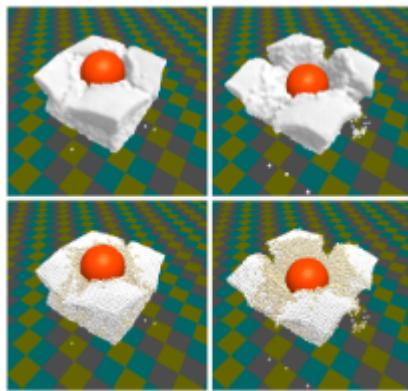
- 目标：模拟场景中雪堆积的效果
- 提出了一套雪堆积过程的模型：雪花不是垂直飘落的，因此一些被遮挡的部分也会有雪堆积，模型描述了不同位置的表面雪堆积量的分布情况。
- 提出了一个稳定模型，来判别一个区域中堆积的雪是否能够保持稳定；若不稳定，将雪移动到下方的稳定区域。

A material point method for snow simulation[2013]



- Disney第一次将mpm方法引入到图形学中的物理仿真
- mpm方法适合于模拟连续材质，在引入elasto-plastic本构模型后能够较好表现雪的塑性和刚性
- 通过混合拉格朗日-欧拉方法这套框架，能够自动处理自碰撞、断裂，能处理大形变

Real-time particle-based snow simulation on the GPU[2019]



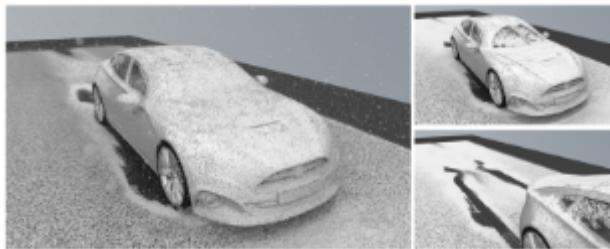
Algorithm 1 Snow Simulation

```

1: while (animating) do
2:   for all particle i do
3:     find neighborhoods  $N_i(t)$ 
4:   for all particle i do
5:     compute  $CohesionForces_i(t)$ 
6:     compute  $Thermodynamics_i(t)$ 
7:     compute  $Compression_i(t)$ 
8:   for all particle i do
9:     update velocity  $\vec{v}(t + \Delta t)$ 
10:    update particle position  $\vec{x}(t + \Delta t)$ 

```

- 能够在GPU并行运算的基于物理的实时仿真方法（计算方法简单，不是真正物理的，根据雪的性质量身定制计算模型）
- 考虑了雪的可压缩性和粘结性(cohesion model)
- 遵循热力学定律(类似SPH热传递) 能处理雪冰相变

An implicit compressible SPH solver for snow simulation[2020]


- 雪的压缩由一种新的压缩压力求解器处理，其中通常使用的状态方程被隐式公式代替。
- 由剪切和法向应力引起的加速度的两个隐式公式的线性求解器是通过无矩阵实现的,有利于稳定性和时间步长。
- 固体边界用粒子表示，并使用一种新的隐式公式来处理固体边界处的摩擦。
- 雪的堆积、变形、破碎、压缩和硬化,与刚体的双向耦合、与不可压缩和高粘性流体的相互作用，从流体到雪的相变。

The Particle-in-Cell Computing Method for Fluid Dynamics.[1964]

PIC方法（质点网格法）是一种较老的混合欧拉-拉格朗日方法，流程步骤如下：

- 根据粒子的分布来更新欧拉网格（Particle to Grid, P2G）
- 在欧拉网格中进行投影（projection）
- 根据欧拉网格的信息来更新粒子的状态（Grid to Particle, G2P）
- 从拉格朗日粒子的角度进行步进（advection）

PIC方法存在的最大问题是：粒子只携带速度信息，其他物理量的信息丢失严重（比如旋转时能量耗散很快）

为了解决信息的丢失问题，人们又提出了两种改进的方法：

- APIC、PolyPIC:让粒子携带更多的信息
- FLIP:传递前后帧的差分或物理量的梯度来代替直接传递值

FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions[1986]

FLIP方法的核心思想是不直接传输物理量的大小，而是改为传递物理量在前后帧的差分或导数。

例如，在PIC方法的G2P步骤中，粒子的速度为周围网格的速度的加权，即：

$$v_p^{t+1} = \text{gather}(v_i^{t+1})$$

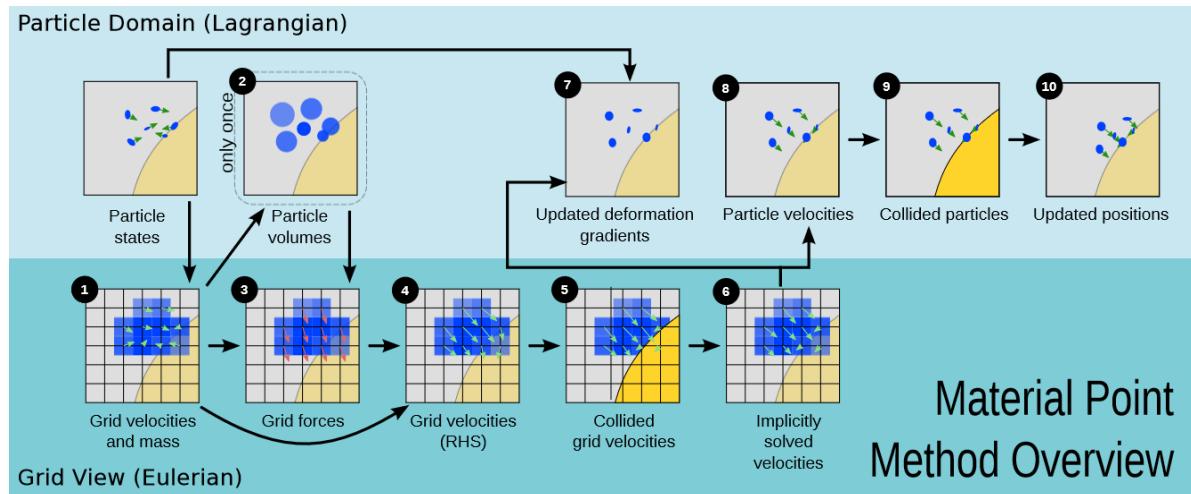
而在FLIP方法则会保留上一帧的粒子和网格信息，G2P仅传递前后帧的差分，即：

$$v_p^{t+1} = v_p^t + \text{gather}(v_i^{t+1} - v_i^t)$$

但FLIP方法模拟的结果会有很大的噪声，一般需要将PIC方法和FLIP方法混合进行模拟。例如FLIPO.99方法：

$$\text{FLIPO.99} = 0.99 * \text{FLIP} + 0.01 * \text{PIC}$$

Application of a particle-in-cell method to solid mechanics[1995]



- 物质点法是一种站在PIC与FLIP方法上的一种混合欧拉-拉格朗日混合方法
- 粒子携带了更多的物理量（密度、体积、质量、形变）
- 相较PIC, advection过程更为复杂（基于粒子的碰撞、基于网格的碰撞）

A material point method for snow simulation中使用的雪的本构模型

Neo-Hookean是最常见的用来预测弹性材料的大型形变的非线性超弹性模型之一。这个模型的能量密度函数为

$$\Psi(\mathbf{F}) = \frac{\mu}{2} (\text{tr}(\mathbf{F}^T \mathbf{F}) - d) - \mu \log(J) + \frac{\lambda}{2} \log^2(J)$$

$d = 2 \text{ or } 3$ 表示问题维度, μ 和 λ 与杨氏模量 E 和泊松率 ν 有关

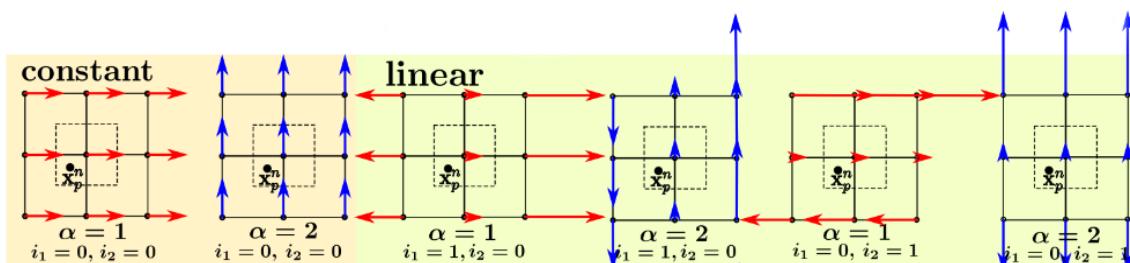
$$\mu = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{Ev}{(1+\nu)(1-2\nu)}.$$

当 F 是旋转的时候, $\psi(F) = 0$ 。对于非反转 F (即 $J > 0$), $\psi(F) \geq 0$ 。

Piola-Kirchoff应力的计算可以根据Neo-Hookean得到 $\mathbf{P} = \frac{\partial \Psi}{\partial F}$

$$\mathbf{P} = \mu (F - F^{-T}) + \lambda \log(J) F^{-T}$$

the affine particle-in-cell method[2016]



- 除了位置和速度, APIC方法为每个粒子增加了一个Affine(仿射)矩阵, Affine矩阵会记录粒子的散度和角速度。

- Affine矩阵从物理意义上讲与形变张量是相似的。
- Apic方法能严格证明角动量守恒，公式证明很复杂，但实现很简单，计算速度快

5.3.1 Particle to grid

Proposition 5.4. Angular momentum is conserved during the APIC transfer from particles to the grid. $\mathbf{L}_{tot}^{G,n} = \mathbf{L}_{tot}^{P,n}$ under transfer 3.

Proof. The angular momentum on the grid after transferring from particles is

$$\begin{aligned}\mathbf{L}_{tot}^{G,n} &= \sum_i \mathbf{x}_i^n \times m_i \mathbf{v}_i^n \\ &= \sum_p \sum_i \mathbf{x}_i^n \times m_p w_{ip}^n (\mathbf{v}_p^n + \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} (\mathbf{x}_i^n - \mathbf{x}_p^n)) \\ &= \sum_p \sum_i \mathbf{x}_i^n \times m_p w_{ip}^n \mathbf{v}_p^n + \sum_p \sum_i \mathbf{x}_i^n \times m_p w_{ip}^n \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_i^n - \sum_p \left(\sum_i w_{ip}^n \mathbf{x}_i^n \right) \times m_p \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_p^n \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p \sum_i w_{ip}^n \mathbf{x}_i^n \times (\mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_i^n) - \sum_p m_p \mathbf{x}_p^n \times (\mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_p^n) \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p \left(\sum_i w_{ip}^n \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_i^n (\mathbf{x}_i^n)^T \right)^T : \epsilon - \sum_p m_p \left(\mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_p^n (\mathbf{x}_p^n)^T \right)^T : \epsilon \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p \left(\sum_i w_{ip}^n \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_i^n (\mathbf{x}_i^n)^T - \mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{x}_p^n (\mathbf{x}_p^n)^T \right)^T : \epsilon \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p \left(\mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \left(\sum_i w_{ip}^n \mathbf{x}_i^n (\mathbf{x}_i^n)^T - \mathbf{x}_p^n (\mathbf{x}_p^n)^T \right) \right)^T : \epsilon \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p \left(\mathbf{B}_p^n (\mathbf{D}_p^n)^{-1} \mathbf{D}_p^n \right)^T : \epsilon \\ &= \sum_p \mathbf{x}_p^n \times m_p \mathbf{v}_p^n + \sum_p m_p (\mathbf{B}_p^n)^T : \epsilon\end{aligned}$$

5.3.2 Grid to particle

Proposition 5.5. Angular momentum is conserved during the APIC transfer from the grid to particles. $\mathbf{L}_{tot}^{P,n+1} = \mathbf{L}_{tot}^{G,n+1}$ under transfer 3.

Proof. As before, the manipulation $(\mathbf{v}\mathbf{u}^T)^T : \epsilon = \mathbf{u} \times \mathbf{v}$ is used to convert the permutation tensor into a cross product. Then on the particles after transferring from the grid is

$$\begin{aligned}\mathbf{L}_{tot}^{P,n+1} &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_p m_p (\mathbf{B}_p^{n+1})^T : \epsilon \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_p m_p \left(\sum_i w_{ip}^n \tilde{\mathbf{v}}_i^{n+1} (\mathbf{x}_i^n - \mathbf{x}_p^n)^T \right)^T : \epsilon \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_p m_p \sum_i w_{ip}^n (\mathbf{x}_i^n - \mathbf{x}_p^n) \times \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_p m_p \sum_i w_{ip}^n \mathbf{x}_p^n \times \tilde{\mathbf{v}}_i^{n+1} + \sum_p m_p \sum_i w_{ip}^n \mathbf{x}_i^n \times \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} - \sum_p m_p \mathbf{x}_p^n \times \sum_i \tilde{\mathbf{v}}_i^{n+1} + \sum_i \left(\sum_p m_p w_{ip}^n \right) \mathbf{x}_i^n \times \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} - \sum_p m_p \mathbf{x}_p^n \times \sum_i w_{ip}^n \tilde{\mathbf{v}}_i^{n+1} + \sum_i m_i^n \mathbf{x}_i^n \times \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_p m_p \mathbf{x}_p^n \times \mathbf{v}_p^{n+1} + \sum_i \mathbf{x}_i^n \times m_i^n \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \mathbf{x}_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \sum_i \mathbf{x}_i^n \times m_i^n \tilde{\mathbf{v}}_i^{n+1} \\ &= \sum_p \Delta t v_p^{n+1} \times m_p \mathbf{v}_p^{n+1} + \mathbf{L}_{tot}^{G,n+1} \\ &= \mathbf{L}_{tot}^{G,n+1}\end{aligned}$$

A massively parallel and scalable multi-GPU material point method 【2020】

作者用现代多GPU架构的强大功能，作者提出了一个基于材料点法的大规模并行仿真系统，用于模拟经历复杂拓扑变化、自碰撞和大变形的材料物理行为。首先，作者引入了一种新的粒子数据结构，该结构促进了GPU上的合并内存访问模式，并消除了将粒子数据写入网格时对内存层次结构进行复杂原子操作的需要。其次，作者提出了一种使用新的网格到粒子到网格（G2P2G）方案的kernel融合方法，该方法有效地减少了GPU kernel的启动，改善了latency，并显着减少了存储粒子数据所需的全局内存量。最后，作者介绍了优化的算法设计，允许在共享内存上下文中使用高效的稀疏网格，能够最好地利用现代多GPU计算平台进行混合拉格朗日-欧拉计算模式。

研究内容

使用《A material point method for snow simulation》中提出的算法流程模拟雪的现象。

算法流程包括：

1. P2G (质量+速度)
2. 第一次update时，根据初始粒子的分布情况计算粒子的体积和密度
3. 根据粒子的形变梯度矩阵和外力计算背景网格上的作用力
4. 根据外力更新背景网格的速度
5. 处理背景网格和刚体、地面的碰撞，更新背景网格的速度
6. 根据背景网格的速度更新粒子的形变梯度矩阵
7. G2P 使用PIC和FLIP方法的混合来更新粒子的速度
8. 处理粒子和刚体、地面的碰撞，更新粒子的速度
9. 更新粒子的位置

模拟的现象包括：

- 文字形状的雪下落堆积
- 抛掷立方体形状的雪
- 抛掷球形的雪
- 平面推雪
- 立方体雪落到雪地上
- 球形刚体落地雪地上
- 两个立方体的雪在空中碰撞

模拟的性质包括：

- 堆积
- 断裂

- 破碎

此外，用非物理方法在unity中实现一小车在雪地行驶的场景，根据小车的运动情况能在雪地上留下不同的痕迹。该算法流程包括：

1. 用一张texture记录雪地的深度
2. 小车的车轮每一次update时更新texture中对应位置的深度值
3. texture中的深度值会随着时间恢复
4. 渲染时，vertex shader中按照texture中的深度值进行顶点位置偏移；fragment shader中按照深度值将地面颜色和雪的颜色作混合

实验方法

1. 在taichi中进行三维可视化
 - 能够键鼠修改相机视角，便于观察
 - 使用particle、mesh等接口绘制雪粒子、地面、平面、刚体等对象
2. 设计系统，各个组件解耦
 - Config模块是配置参数，通过data_oriented修饰后保证参数在cpu上可读可写，在gpu上可读

```
@ti.data_oriented
class Config:
    def __init__(self):
        self.frameTime=0.016
        self.stepsPerFrame=40
        self.maxParticles:int=100000
        self.gridSize:float=0.1
        self.gridNumX:int=160
        self.gridNumY:int=80
        self.gridNumZ:int=160
        self.hardening_coefficient:float=10.0
        self.poissos_ratio:float = 0.2
        self.youngs_modulus:float = 1.4e5
        self.mu:float=self.youngs_modulus/(2.0*
(1.0+self.poissos_ratio))

        self.lam:float=self.youngs_modulus*self.poissos_ratio/((1.0+self.poissos_ratio)* (1 - 2 * self.poissos_ratio))

        self.filp_alpha:float=0.95
        self.friction_coeff:float=0.35
        self.critical_compression:float=0.025
        self.critical_stretch:float=0.0075
```

- Simulation模块负责场景模拟，update函数和render函数是公有的。不同的场景继承自基类，并重载init函数，初始化不同场景中的不同物体

```
@ti.data_oriented
class Simulation:
    def __init__(self,config:Config):
        self.groundManager=GroundManager(config)
```

```

self.rigidBodyManager=RigidBodyManager(self.groundManager,config)

self.particleManager=ParticleManager(self.rigidBodyManager,self.groundManager,config)
    self.config:Config=config


def update(self):
    dt=self.config.frameTime

    for x in range(self.config.stepsPerFrame):
        self.groundManager.step(dt/self.config.stepsPerFrame)
        self.rigidBodyManager.step(dt/self.config.stepsPerFrame)
        self.particleManager.step(dt/self.config.stepsPerFrame)

def render(self,scene:ti.ui.Scene):
    self.groundManager.render(scene)
    self.rigidBodyManager.render(scene)
    self.particleManager.render(scene)

```

- o Ground Manager模块负责处理地面和平面的碰撞
- o Rigid Manager模块负责处理刚体的碰撞
- o Particle Manager是最核心的模块，包含了mpm计算流程的各个步骤

```

@ti.data_oriented
class ParticleManager:
    def __init__(self,rigidBodyManager,groundManager,config:Config):
        self.rigidBodyManager : RigidBodyManager=rigidBodyManager
        self.groundManager:GroundManager=groundManager
        self.particlesNum=0

        #particle
        self.pos=ti.Vector.field(3,dtype=float,shape=config.maxParticles)
        self.vel=ti.Vector.field(3,dtype=float,shape=config.maxParticles)
        self.mass=ti.field(float,shape=config.maxParticles)
        self.volume=ti.field(float,shape=config.maxParticles)
        self.density=ti.field(float,shape=config.maxParticles)

        self.elastic=ti.Matrix.field(3,3,dtype=float,shape=config.maxParticles)

        self.plastic=ti.Matrix.field(3,3,dtype=float,shape=config.maxParticles)

        #grid
        gridNum=config.gridNumX*config.gridNumY*config.gridNumZ
        self.gridOldVelocity=ti.Vector.field(3,dtype=float,shape=gridNum)
        self.gridVelocity=ti.Vector.field(3,dtype=float,shape=gridNum)
        self.gridForce=ti.Vector.field(3,dtype=float,shape=gridNum)
        self.gridMass=ti.field(float,shape=gridNum)

        # others
        self.firstIteration=True
        self.config:Config=config
        self.textdata=ti.field(float,11*38)

```

```

        for i in range(11*38):
            self.textdata[i]=data[i]

    def step(self,dt):
        self.clearCache()
        self.rasterizeParticles()
        if self.firstIteration:
            self.calVolumes()
        self.firstIteration=False
        self.calculateForces()
        self.updateGridVelocity(dt)
        self.handleGridBasedCollision(dt)
        self.updateDeformationGradient(dt)
        self.updateParticleVelocity()
        self.handleParticleBasedCollision(dt)
        self.updateParticlePosition(dt)

    def render(self,scene:ti.ui.Scene):
        scene.particles(self.pos, radius=0.05, color=(0.9, 0.9,
        0.9), index_count=self.particlesNum)

        .....

```

3. mpm的基本流程

论文中对于算法步骤的描述比较详细，此外还有16年的siggraph tutorial作为补充

```

def step(self,dt):
    self.clearCache()
    self.rasterizeParticles()
    if self.firstIteration:
        self.calVolumes()
    self.firstIteration=False
    self.calculateForces()
    self.updateGridVelocity(dt)
    self.handleGridBasedCollision(dt)
    self.updateDeformationGradient(dt)
    self.updateParticleVelocity()
    self.handleParticleBasedCollision(dt)
    self.updateParticlePosition(dt)

```

按照步骤把公式先敲完再说，不管写的对不对，确保没有运行时错误。

4. 性能优化

程序跑的太慢了，会浪费调试的时间。

比如说，把雪球跑出去这个场景，雪球在空中的时候形变梯度没什么变化的，要等雪球和地面接触后才能观察到异常。程序运行的快，那等待时间就少了

优化方式：

- 参照官网文档中的优化指南检查自己的写法
- 增加程序的并行性，程序片段中避免有data race风险的同一变量又读又写
- 计算各个步骤的运行时间，发现异常的代码片段（比如在gpu上动态创建大的数组，gpu中频繁往cpu写数据）

5. 分析代码哪里写的不对

- 单元测试

如P2G、G2P计算相邻网格的权重时,按照插值公式, 这些权重之和应该在1附近, 如果算出来的和不对, 说明公式就敲的有问题

```
for a,b,c in ti.ndrange(4,4,4):
    grid_x=gridIndexX+a-1
    grid_y=gridIndexY+b-1
    grid_z=gridIndexZ+c-1
    offsetX=posX-dx*grid_x
    offsetY=posY-dx*grid_y
    offsetZ=posZ-dx*grid_z
    weight=calGridWeight(offsetX,offsetY,offsetZ,idx)
```

- 异常数值追踪

比如, 模拟的现象中出现了这么一个现象, 运行着运行着一个立方体区域内的粒子消失了。

在排除数值变为Nan的可能性后, 初步推断是update中速度的变化值过大导致的。

速度变化过大->gridForce异常->形变梯度异常->sigma数值异常->mu、lambda数值异常->plastic形变梯度矩阵的行列式过小

由于时间步长还是比较大的, plastic_determinant过小在迭代中会出现数值爆炸现象。

处理方法: 对行列式作截断, 也许可以解释为约束粒子的弹性形变范围。

像这种bug就属于比较难调的了, 要靠经验去猜出问题的原因。

```
plastic_determinant=self.plastic[x].determinant()
if plastic_determinant<0.5:
    plastic_determinant=0.5
elastic_determinant=self.elastic[x].determinant()
RE, SE = ti.polar_decompose(self.elastic[x])
mu=self.config.mu*ti.exp(self.config.hardening_coefficient*(1.0-
plastic_determinant))
lam=self.config.lam*ti.exp(self.config.hardening_coefficient*(1.0-
plastic_determinant))
sigma=2.0*mu*(self.elastic[x]-RE)@self.elastic[x].transpose() + lam*
(elastic_determinant-1.0)*elastic_determinant*ti.Matrix.identity(float,3)
```

6. 设计不同的场景

```
@ti.kernel
def init(self):
    # 添加平面
    self.groundManager.addGround(8, 8, 16)

    # 添加雪的粒子
    for i in range(11):
        for j in range(38):
            for k in range(6):
                for m in range(10):
                    x=i*38*60+j*60+k*10+m
                    if(self.particleManager.textdata[j*11+i]==1):
```

```

        self.particleManager.pos[x]=
[3.0+k*0.1+0.1*ti.random(float),3.0-0.1*i+0.1*ti.random(float),12.0-
0.1*j+0.1*ti.random(float)]
    else:
        self.particleManager.pos[x]=[-100,0.0,-100]
self.particleManager.vel[x]=[0.0,0.0,0.0]
self.particleManager.density[x]=0
self.particleManager.volume[x]=0
self.particleManager.mass[x]=0.2

self.particleManager.plastic[x]=ti.Matrix.identity(float,3)

self.particleManager.elastic[x]=ti.Matrix.identity(float,3)

```

如上，在不同的Simulation的init函数里面设计场景

7. 针对不同的场景调参数

实验结果和验证

实验平台

操作系统: windows

GPU: 3080ti

编程框架: taichi

后端: cuda

基本参数配置

硬度系数 hardening_coefficient = 10， 它会影响网格上由粒子形变梯度矩阵引起的受力大小

泊松率 poissons_ratio = 0.2

杨氏模量 youngs_modulus = 1.4e5

FLIP混合系数 flip_alpha = 0.95

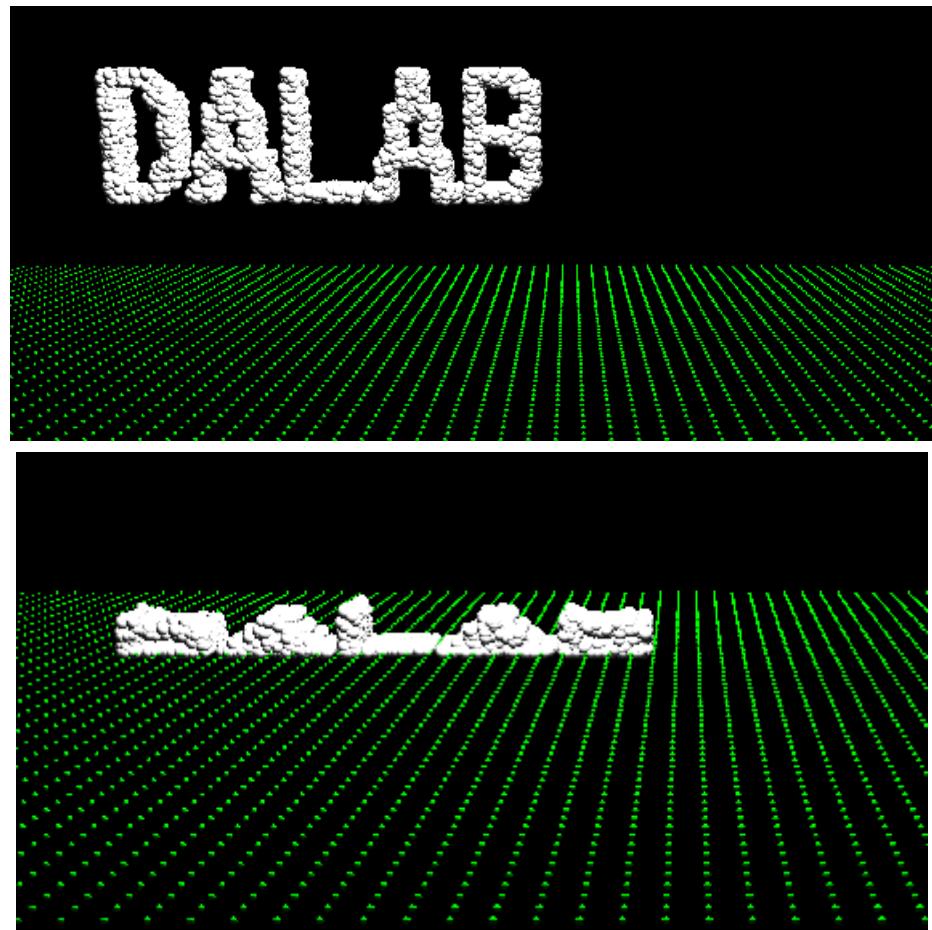
地面和雪的摩擦系数 friction_coeff = 0.35

形变的压缩截断 critical_compression = 0.025

形变的拉伸截断 critical_stretch = 0.0075

结果

- 文字形状的雪下落堆积

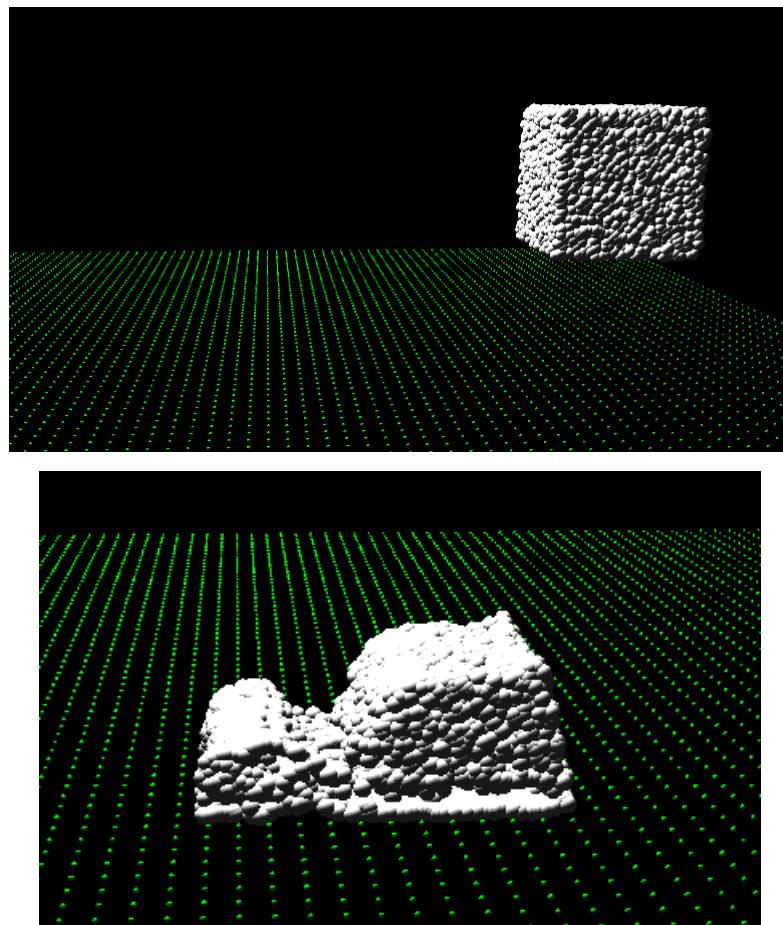


- 抛掷立方体形状的雪

为了实现断裂的效果，在基础配置上修改了如下两项

hardening_coefficient=25.0

filp_alpha=0.98

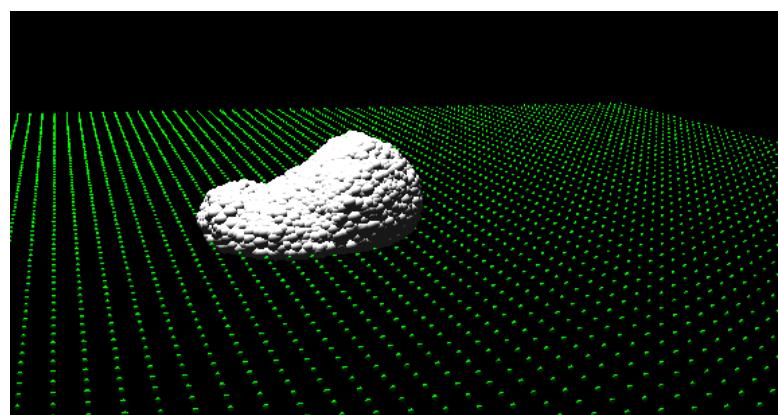
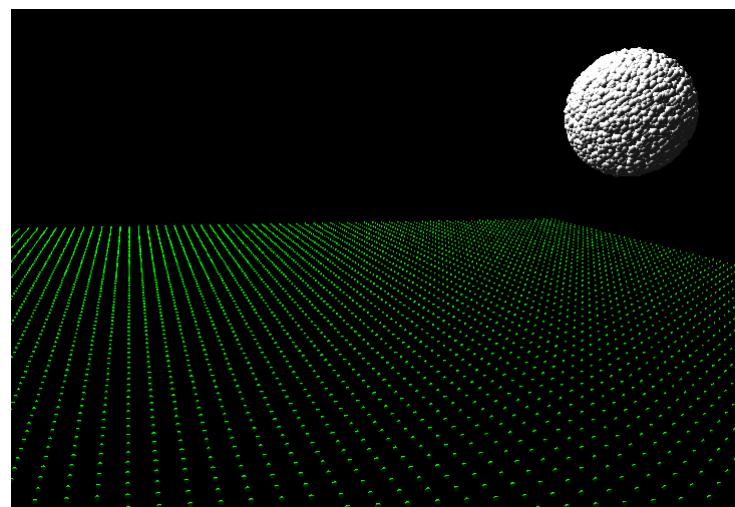


- 抛掷球形的雪

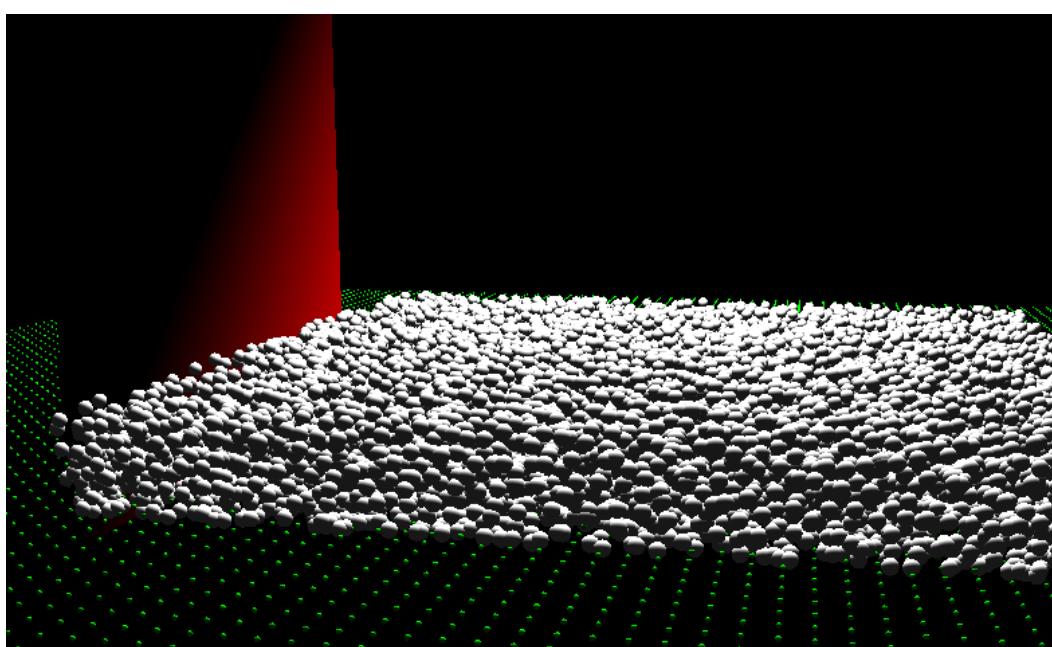
为了增大落地后的形变，作以下调整

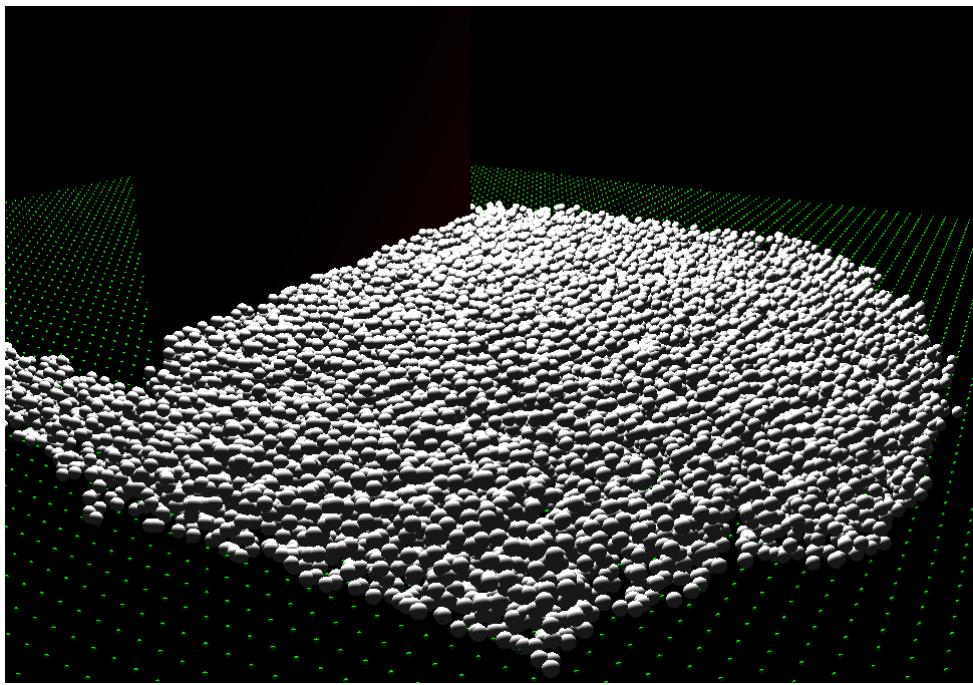
hardening_coefficient=25.0

filp_alpha=0.98



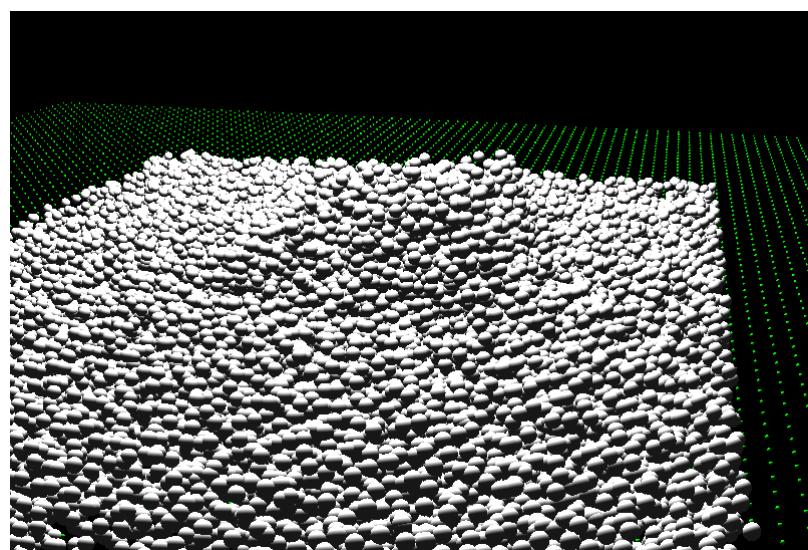
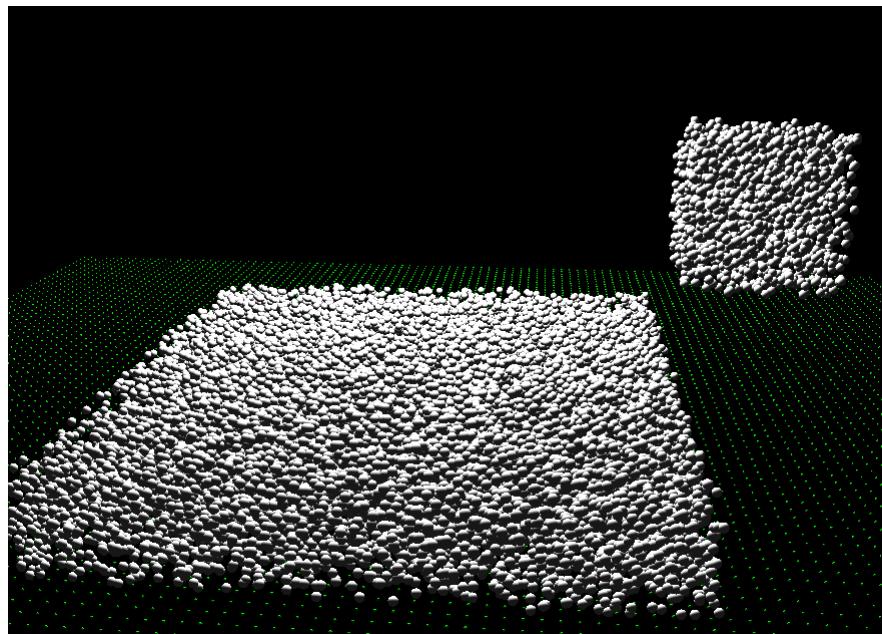
- 平面推雪



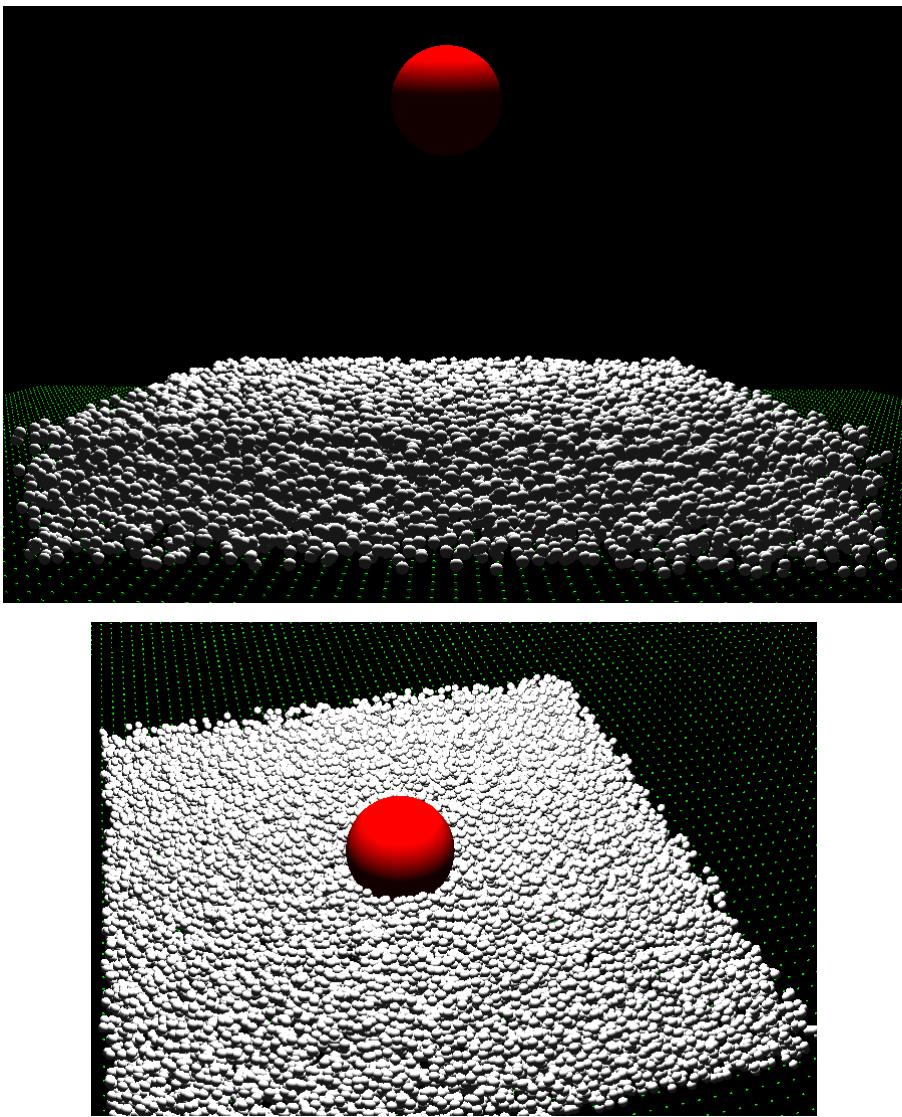


- 立方体雪落到雪地上

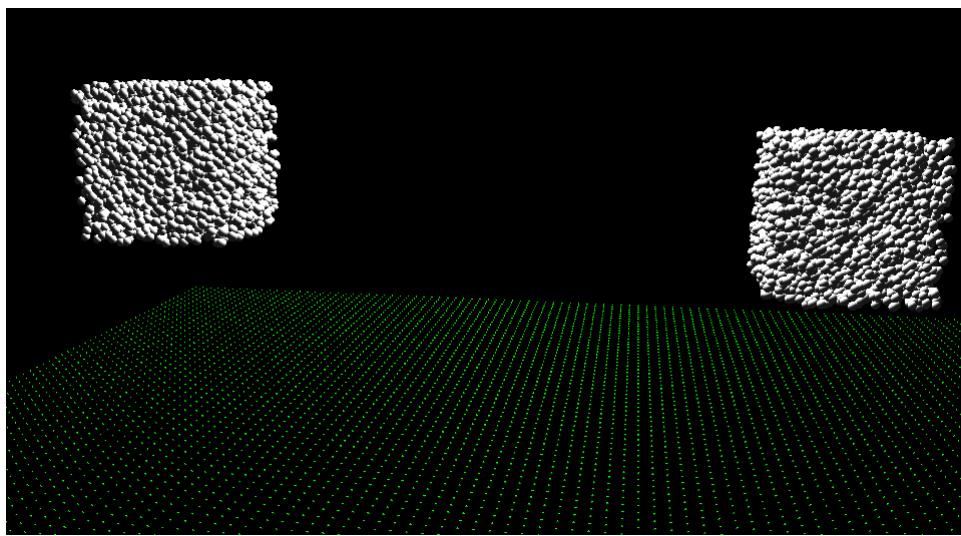
为增大雪掉落到雪地上的形变程度，提高hardening_coefficient至12.0

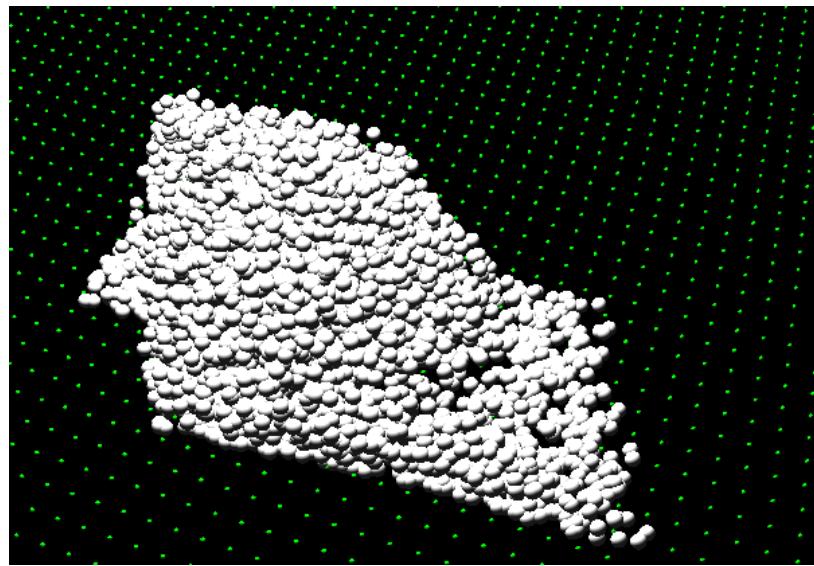


- 球形刚体落地雪地上



- 两个立方体的雪在空中碰撞





性能

网格规模: 160 * 80 * 160

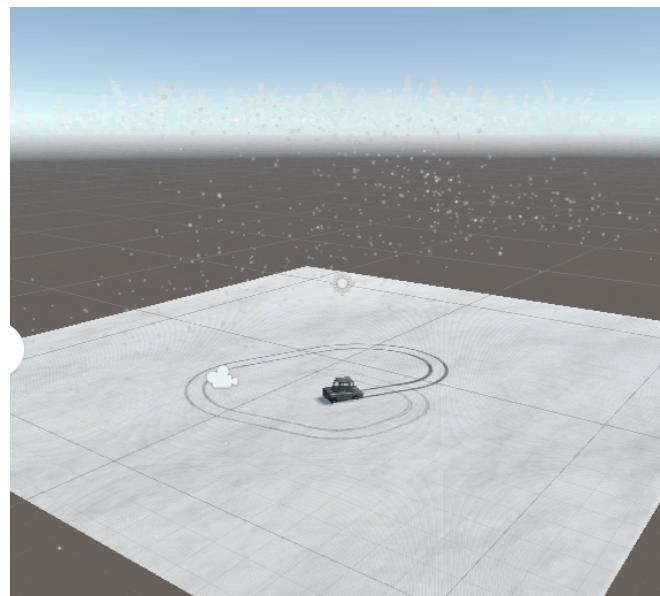
粒子数量: 20000 +

时间步长: 1/60

每个步长中的迭代次数: 40

画面帧率: ~ 34 frame/s

unity中的模拟



小车在雪地上行驶

项目小结

总结

用taichi实现雪的模拟这个目标基本算是实现了，历时断断续续超过一个月。

项目中最大的难点是把mpm写对，代码中隐藏的错误（可能不算是错误，而是缺少工程trick）实在是不好找。

项目中最大的收获是对于混合欧拉网格方法的思想有了深入的理解，希望有一天能在别的地方用到。

项目中最大的遗憾是雪还是不够逼真。参数调的不够好是一小部分原因，另一方面我觉得是代码只实现了基本方法，模型还是不够复杂，工程细节上的处理不够到位。现在只实现了基本流程，按照对现有参数的调试经验，我觉得没法做到“两个雪球碰撞时产生大小不一的碎块”。

优缺点

优点：

- mpm基本实现对了
- 把性能优化到了实现
- 设计了比较多的场景，不同场景中的参数基本是一致的，证明了算法能自然地表现破碎、断裂、堆积、粘滞等雪的特性
- 比较了物理仿真和游戏仿真中实现雪的截然不同的思路

缺点：

- 不够逼真，和论文里渲染的效果没法比
- 没有复杂的场景，在taichi中导入外部的刚体模型以及运算比较麻烦
- 瓶颈是背景网格数量，局限了仿真的空间大小，按照现有的写法，没法做大场景的仿真

展望

雪的模拟算是告一段落了，希望能把mpm方法或者思想创新地使用在另一个领域。

参考文献

- [1] Sulsky D, Zhou S J, Schreyer H L. Application of a particle-in-cell method to solid mechanics[J]. Computer physics communications, 1995, 87(1-2): 236-252.
- [2] Stomakhin A, Schroeder C, Chai L, et al. A material point method for snow simulation[J]. ACM Transactions on Graphics (TOG), 2013, 32(4): 1-10.
- [3] Li X, Cao Y, Li M, et al. Plasticitynet: Learning to simulate metal, sand, and snow for optimization time integration[J]. Advances in Neural Information Processing Systems, 2022, 35: 27783-27796.
- [4] Gissler C, Henne A, Band S, et al. An implicit compressible SPH solver for snow simulation[J]. ACM Transactions on Graphics (TOG), 2020, 39(4): 36: 1-36: 16.
- [5] Goswami P, Markowicz C, Hassan A. Real-time particle-based snow simulation on the GPU[C]//EGPGV 2019. Eurographics-European Association for Computer Graphics, 2019.
- [6] Tskhakaya D, Matyash K, Schneider R, et al. The Particle-In-Cell Method[J]. Contributions to Plasma Physics, 2007, 47(8-9): 563-594.
- [7] Brackbill J U, Ruppel H M. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions[J]. Journal of Computational physics, 1986, 65(2): 314-343.
- [8] Wang X, Qiu Y, Slattery S R, et al. A massively parallel and scalable multi-GPU material point method[J]. ACM Transactions on Graphics (TOG), 2020, 39(4): 30: 1-30: 15.
- [9] Nishita T, Iwasaki H, Dobashi Y, et al. A modeling and rendering method for snow by using metaballs[C]//Computer Graphics Forum. Oxford, UK and Boston, USA: Blackwell Publishers Ltd, 1997, 16(3): C357-C364.
- [10] Fearing P. Computer modelling of fallen snow[C]//Proceedings of the 27th annual conference on Computer graphics and interactive techniques. 2000: 37-46.