Position Based Fluid

题目介绍

Position Based Fluid 是一种使用粒子系统模拟液体的方法，在 2013 年由 Macklin 等人提出，发表在 SIGGRAPH 上。核心思想是利用液体的不可压缩性，根据粒子的空间位置和密度来更新每一个粒子的位置，达到模拟的效果。不同于传统力学模拟，这种方法只使用了几何位置约束，不存在逐步积分带来的累积误差。另外，这种方法能够高度并行化，加速模拟计算，做到实时模拟、渲染、显示。

我做的工作是实现这篇论文中提出的模拟方法。我使用的是 OpenGL 渲染，CUDA9.2 做并行计算，求解约束。最后实现了几个用于演示的 demo 情景。

论文回顾

不可压缩性的实现(Enforcing Incompressibility)

保证液体的不可压缩性需要求解一系列的非线性约束，系统中的每一个粒子代表一个约束。首先定义系统中每一个粒子的密度，根据 SPH 方法，系统中第 i 个粒子的密度定义如下：

$$\rho_i = \sum_j m_j W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)$$

其中 j 代表这个粒子的所有"邻居"粒子。M 代表质量，p 代表位置。W 是一个 SPH 核，我使用 Poly6 核进行计算，Poly6 核定义见附录。

有了密度，引出我们的约束条件:

$$\boldsymbol{C}_i(\boldsymbol{p}_1, \cdots, \boldsymbol{p}_n) = \frac{\rho_i}{\rho_0} - 1$$

$$\boldsymbol{C}(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0$$

其中 ρ_0 代表液体的预设密度，也叫 rest density。这个常数可以自行设置。P 代表粒子现在的位置，△p 代表粒子的位移。要求解的就是这个位移，要求位移之后密度约束仍然被满足。

论文使用牛顿迭代法求解，推导过程就不叙述了。最后我们需要的方程如下：

$$\Delta\boldsymbol{p}_i = \frac{1}{\rho_0}\sum_j (\lambda_i + \lambda_j)\nabla W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)$$

求解第 i 个粒子的运动我们需要知道他和他"邻居"粒子的：

$$\lambda_i = -\frac{\boldsymbol{C}_i(\boldsymbol{p}_1, \cdots, \boldsymbol{p}_n)}{\sum_k |\nabla_{\boldsymbol{p}_k}\boldsymbol{C}_i|^2 + \epsilon}$$

其中 ε 为常量。

上述两个方程中，约束的梯度我们使用 SPH 中另一个核函数即 spiky 核计算，spiky 核的定义见附录，梯度计算的方程为：

$$\nabla_{\boldsymbol{p}_k} \boldsymbol{C}_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\boldsymbol{p}_k} W(\boldsymbol{p}_i - \boldsymbol{p}_j, h) & if\ k = i \\ -\nabla_{\boldsymbol{p}_k} W(\boldsymbol{p}_i - \boldsymbol{p}_j, h) & if\ k = j \end{cases}$$

张量不稳定性的解决(Tensile Insability)

为了解决在模拟中经常遇到的粒子由于邻居太少，不能满足初始密度而被负压力聚拢的问题，论文提出加入一个人工压力(artificial pressure)，计算方式如下:

$$s_{corr} = -k \left(\frac{W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)}{W(\Delta \boldsymbol{q}, h)} \right)^n$$

将这个量加入计算△p 的公式中，得到：

$$\Delta \boldsymbol{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}) \nabla W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)$$

其中，k\△q\n 均为常量。

涡流限制与粘滞系数的实现(Vorticity Confinement And Viscosity)

使用 Position Based Fluid 计算得到的运动常常会过早的进入稳定状态，即会有比较大的阻尼。为了弥补这一缺点，论文提出了加入涡流限制来弥补损失的能量：
首先对每一个粒子做计算：

$$\boldsymbol{\omega}_i = \nabla \times \boldsymbol{v} = \sum_j \boldsymbol{v}_{ij} \times \nabla_{\boldsymbol{p}_j} W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)$$

之后使用如下方程计算一个修正的涡流力：

$$\boldsymbol{f}_i^{vorticity} = \epsilon(\boldsymbol{N} \times \boldsymbol{\omega}_i)$$

$$\boldsymbol{N} = \frac{\eta}{\|\eta\|} \text{ and } \eta = \Delta \|\boldsymbol{\omega}\|_i$$

最后，论文还使用了 XSPH viscosity(不知道怎么翻译)来保证运动的统一性：

$$\boldsymbol{v}_i^{new} = \boldsymbol{v}_i + c \sum_j \boldsymbol{v}_{ij} \cdot W(\boldsymbol{p}_i - \boldsymbol{p}_j, h)$$

最后总结整个算法流程如下：

Algorithm 1 Simulation Loop

```
for all particles i do
    apply forces v_i ⇐ v_i + Δt f_ext(x_i)
    predict position x_i* ⇐ x_i + Δt v_i
end for
for all particles i do
    find neighboring particles N_i(x_i*)
end for
while iter < solverIterations do
    for all particles i do
        calculate λ_i
    end for
    for all particles i do
        calculate Δp_i
        perform collision detection and response
    end for
    for all particles i do
        update position x_i* ⇐ x_i* + Δp_i
    end for
end while
for all particles i do
    update velocity v_i ⇐ 1/Δt (x_i* − x_i)
    apply vorticity confinement and XSPH viscosity
    update position x_i ⇐ x_i*
end for
```

代码实现

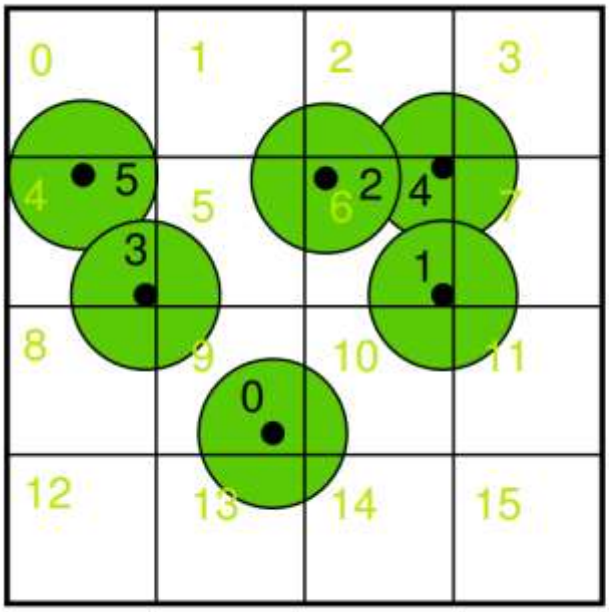
粒子系统

首先肯定要有一个粒子系统来做这样的物理模拟。我曾经写过一个 CPU 版的小粒子系统来做 OpenGL 的特效，最开始我尝试将那个粒子系统直接搬过来。但发现由于这样的模拟需要每一次循环都重新计算每一个粒子的邻居，而在 CPU 上这样的操作相当花时间，因此这个方案很快就被放弃了。

之后我参考了 CUDA 9.2 自带的 Sample 与 Particle Simulation using CUDA[1]。编写了一个基于 CUDA 的粒子系统，能够很好的利用 GPU 的并行特性，很快的解决给每一个粒子寻找邻居这样一个消耗巨大的操作。

具体的做法是：将整个三维空间划分为统一大小的三维格子，给每一个格子根据坐标赋予一个哈希值，利用哈希表存储每一个格子中的粒子。这样在寻找邻居的时候，只要根据粒子的坐标计算出所属的格子，之后顺序查找周边 3*3*3 共 27 个格子就可以完成一次邻居搜索。显然，这样的操作可以并行化，即建立哈希表和寻找邻居分别可以以一个粒子为单位完全并行操作，如下图所示：

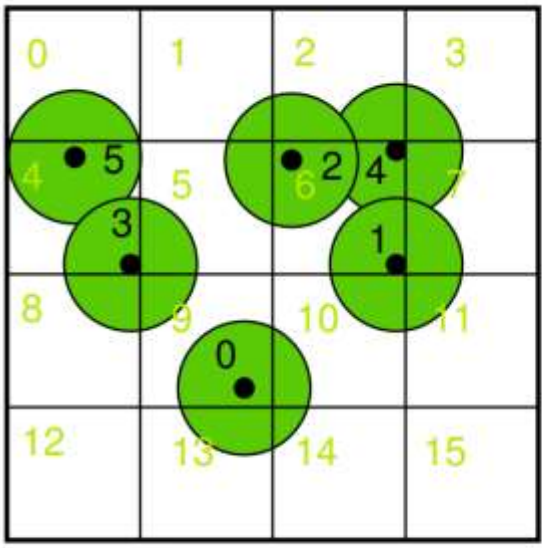
[1] By Simon Green May 2010



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

关于邻居粒子的存储，文章中给出了两种解决方案：

一种是根据排序，将哈希表用格子的哈希值与其中的粒子编号排序，用两个数组保存每一个格子的起点和终点坐标，要使用的时候进行查找，这也是 CUDA Sample 中使用的方法，如下图：



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

另一种是给每一个格子和每一个粒子建立一张单独的哈希表，其中每个格子和每个粒子都会有一个存储上限。以粒子的邻居表为例：若有 n 个粒子，规定粒子邻居数目不超过 50 个，这样邻居表的大小就是 n * 50，每个粒子根据自己的编号有一个偏移量，存储自己的邻居。这样不需要另外的数组保存每一个格子的起点和终点。这种方法在统计每个格子的粒子数目的时候需要用到原子加法保证统计的准确。

我两种方法都实现了，但最后选择了第二种方法实现 PBF。主要原因是：PBF 求解过程中，每个粒子的邻居数不会很大，第二种方法在实现的简便性和存取邻居的方便性上更占优势。

PBF 求解并行化

PBF 方法本身就具有很好的并行性，适合用 GPU 做加速，回顾求解的伪代码：

Algorithm 1 Simulation Loop

```
for all particles i do
    apply forces v_i ⇐ v_i + Δt f_ext(x_i)
    predict position x_i* ⇐ x_i + Δt v_i
end for
for all particles i do
    find neighboring particles N_i(x_i*)
end for
while iter < solverIterations do
    for all particles i do
        calculate λ_i
    end for
    for all particles i do
        calculate Δp_i
        perform collision detection and response
    end for
    for all particles i do
        update position x_i* ⇐ x_i* + Δp_i
    end for
end while
for all particles i do
    update velocity v_i ⇐ (1/Δt)(x_i* − x_i)
    apply vorticity confinement and XSPH viscosity
    update position x_i ⇐ x_i*
end for
```

其中，第一步根据速度和外力更新位置与速度显然可以并行化。

第二步根据更新后的位置重新生成邻居表如上文所述，可以并行化，代码调用顺序如下：

```
clearCells <<< grid_numBlocks, numThreads >>> (
    cells_count
    );
clearNeighbors <<< numBlocks, numThreads >>> (
    neighbors_count
    );
updateCells <<< numBlocks, numThreads >>> (
    (float4*)newPos,
    cells,
    cells_count
    );
updateNeighbors <<< numBlocks, numThreads >>> (
    (float4*)newPos,
    cells,
    cells_count,
    neighbors,
    neighbors_count
    );
```

具体说来就是首先清空上次保存下来的格子与邻居信息，之后先并行更新格子中包含的粒子信息，在并行更新每一个粒子的邻居信息。

第三步迭代求解△p，每一个”for all particles I do”的循环体都可以并行化，调用代码如下：

```
for (unsigned int i = 0; i < iters; i++) {
		// get each particle's C(density)
		getDensityD <<< numBlocks, numThreads >>>(
			(float4 *)newPos,
			neighbors,
			neighbors_count,
			particleDensity
			);
		// get each particle's Lamda
		getLamdaD <<< numBlocks, numThreads >>> (
			(float4 *)newPos,
			neighbors,
			neighbors_count,
			particleDensity,
			particleLamda
			);
		// get each particle's fixed position delta-p
		getDpD <<< numBlocks, numThreads >>>(
			(float4 *)newPos,
			(float4 *)particleDeltaPos,
			neighbors,
			neighbors_count,
			particleLamda
			);
		updateNewPositionD <<< numBlocks, numThreads >>> (
			(float4 *)particleDeltaPos,
			(float4 *)newPos);
		confinePositionD <<< numBlocks, numThreads >>>(
			(float4 *)newPos
			);
	}
```

具体来说就是首先并行更新密度，之后根据密度并行计算出 Lambda，再根据 Lambda 与邻居信息并行计算得到每一个粒子的偏移，更新位置并进行碰撞检测也都可以并行化。

最后的更新位置与速度也可以并行化，计算更新后的速度，计算涡流与粘滞速度与最后的速度赋值都可以并行化。

```
	updateVel <<< numBlocks, numThreads >>>(
		(float4 *)oldPos,
		(float4 *)newPos,
		(float4 *)Vel
		);

	calculateOmega <<< numBlocks, numThreads >>> (
		(float4 *)newPos,
		(float4 *)Vel,
		(float3 *)omega,
		neighbors,
		neighbors_count
		);

	getLastCudaError("Kernel execution failed");
	updateVelocity <<< numBlocks, numThreads >>> (
		(float4 *)oldPos,
		(float4 *)newPos,
		(float3 *)omega,
		neighbors,
		neighbors_count,
		(float4 *)Vel,
		particleDensity
		);
```

实现上的困难与解决方法

粒子系统的模拟是十分依赖于参数，且十分不稳定的，即便论文称他的做法十分稳定。以下就是我在几百次的粒子爆炸中总结出的提高系统稳定性，增加模拟真实度的一些做法与技巧。

小球绘制

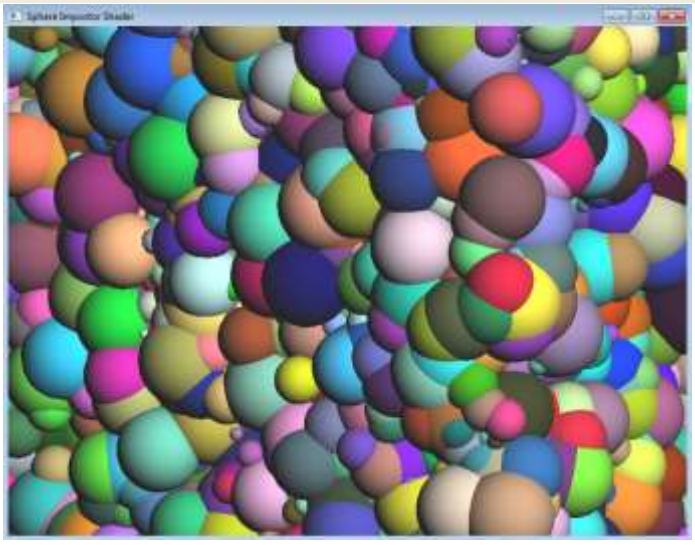
粒子系统最重要的就是执行效率，无论是模拟计算还是绘制。要即时绘制成千上万的粒子并且保证不被人眼看出小球的棱角，使用 mesh 为基础的球体模型肯定不是一个聪明的解决方案。通过查找资料，我找到了利用 OpenGL 的 Point Sprite 来进行小球绘制的方法。其核心就是利用 Fragment Shader 的渲染将一个长方形渲染成一个圆形，并利用 Phong Lighting Model 渲染出阴影，欺骗人眼，让人以为那是一个小球。Shader 代码如下：

```
#version 400

out vec4 frag_colour;

float Ns = 250;
vec4 mat_specular=vec4(1);
vec4 light_specular=vec4(1);
const float PI = 3.14159265358979323846264338327795;

void main()
{
	vec3 lightDir = vec3(0.3,0.3,0.9);
	// calculate normal from texture coordinates
	vec3 N;
	N.xy = gl_PointCoord* 2.0 - vec2(1.0);
	float mag = dot(N.xy, N.xy);
	if (mag > 1.0) discard;	// kill pixels outside circle
	N.z = sqrt(1.0-mag);
	// calculate lighting
	float diffuse = max(0.0, dot(lightDir, N));
	//vec3 eye = vec3 (0.0, 0.0, 1.0);
	//vec3 halfVector = normalize( eye + lightDir);
	//float spec = max( pow(dot(N,halfVector), Ns), 0.);
	//vec4 S = light_specular*mat_specular* spec;
	frag_colour = vec4(0.0f, 0.6235f, 1.0f, 1.0f)*diffuse;
};
```



From: http://11235813tdd.blogspot.com/2013/04/raycasted-spheres-and-point-sprites-vs.html

边界处理

粒子系统模拟中，边界处理一直是比较让人头疼的问题。最开始我是简单的采用完全非弹性碰撞模型进行粒子与边界的碰撞模拟：

```
__global__
void updatePositionD(
	float4* oldPos,
	float4* newPos,
	float4* velocity
)
{
	uint index = blockIdx.x * blockDim.x + threadIdx.x;
	if (index >= params.numBodies) return;

	float3 pos = make_float3(oldPos[index]);
	float3 vel = make_float3(velocity[index]);

	vel += params.gravity * params.deltaTime;
	//vel *= params.globalDamping;

	// new position = old position + velocity * deltaTime
	pos += vel * params.deltaTime;

	if (pos.x < 0) {
		vel.x = 0;
		pos.x = 0.001f;
	}
	else if (pos.x > params.worldbound.x) {
		vel.x = 0;
		pos.x = params.worldbound.x - 0.001f;
	}

	if (pos.y < 0) {
		vel.y = 0;
		pos.y = 0.001f;
	}
	else if (pos.y > params.worldbound.y) {
		vel.y = 0;
		pos.y = params.worldbound.y - 0.001f;
	}

	if (pos.z < 0) {
		vel.z = 0;
		pos.z = 0.001f;
	}
	else if (pos.z > params.worldbound.z) {
		vel.z = 0;
		pos.z = params.worldbound.z - 0.001f;
	}

	newPos[index] = make_float4(pos, 1.0f);
	velocity[index] = make_float4(vel, 0.0f);
}
```

但是这样的做法有严重的弊端：由于粒子系统的更新是离散的，会出现有同一时刻多个粒子挤到同一个角落的情况，当粒子数增加，时间步长增大的时候，这样会造成严重的结果——粒子系统爆炸——以角落为源，粒子由于简单粗暴的被边界推回同一个位置并失去了速度，在下一次求解的时候就会被以极快的速度弹射出去，进而影响整个粒子系统的稳定性。

参考 Position Based Dynamic[2]中的做法，我给边界也赋予了一定的密度值，加入密度求解中，这样能够让粒子不会轻易的靠近边界，减少上述情况发生的可能性；同时减小模拟的步长，增加连续性；关于和边界的碰撞仍然延续之前的做法。

CFL Condition

参数设置的再精巧也总会发生个别粒子由于局部的不稳定，突然加速的情况，为了最大限度的缓解这种情况，我加入了一个最大速度的限制。当然，这个做法也不是我第一个提出来的：参考 Nvidia Flex 的手册，里面提到了液体模拟时需要满足 CFL 条件，之后我才知道有这么一个约束，参考 Flex 的手册，设定如下：

```
// limit velocity to CFL condition
g_params.maxSpeed = 0.5f*radius*g_numSubsteps / g_dt;
```

CFL 条件在模拟中代表了一个 timestep 中粒子(任何信息)能够运动(传播)的最大长度，即只能运动到相邻的区域，对于粒子来说，就是邻居的位置。

参数调整

一切的计算机物理模拟都逃不开”调参”两个字。粒子系统尤其如此。尽管是一篇比较先进且简洁的论文，其中的参数仍然多的让人头疼。从密度计算时的 rest Density，lambda 计算时的松弛系数到后来改进的人工压力的参数、涡流和粘滞性参数都需要细心的调整。更让人头疼的是，这些参数都相互影响，即调了一个参数，相当于调整了所有的参数。如果参数没有调整好，整个系统会相当的不稳定。我也从网络上各个地方找了很多参数进行尝试、调整。最后总结出了我正在使用的，相对较好的一套参数，具体还请参见源代码。

另外，当系统非常不稳定并且参数调整难以起作用的时候，缩短 timestep 永远是一个值得一试的方案。

OpenGL 画面录制

使用 glReadPixels()进行画面采集，用 ffmpeg.exe 录制。具体参考论文作者 Miles Macklin 的博文 http://blog.mmacklin.com/2013/06/11/real-time-video-capture-with-ffmpeg/

[2] MULLER ¨ , M., HEIDELBERGER, B., HENNIX, M., AND RATCLIFF, J. 2007. Position based dynamics. J. Vis. Comun. Image Represent. 18, 2 (Apr.), 109–118.

实验结果

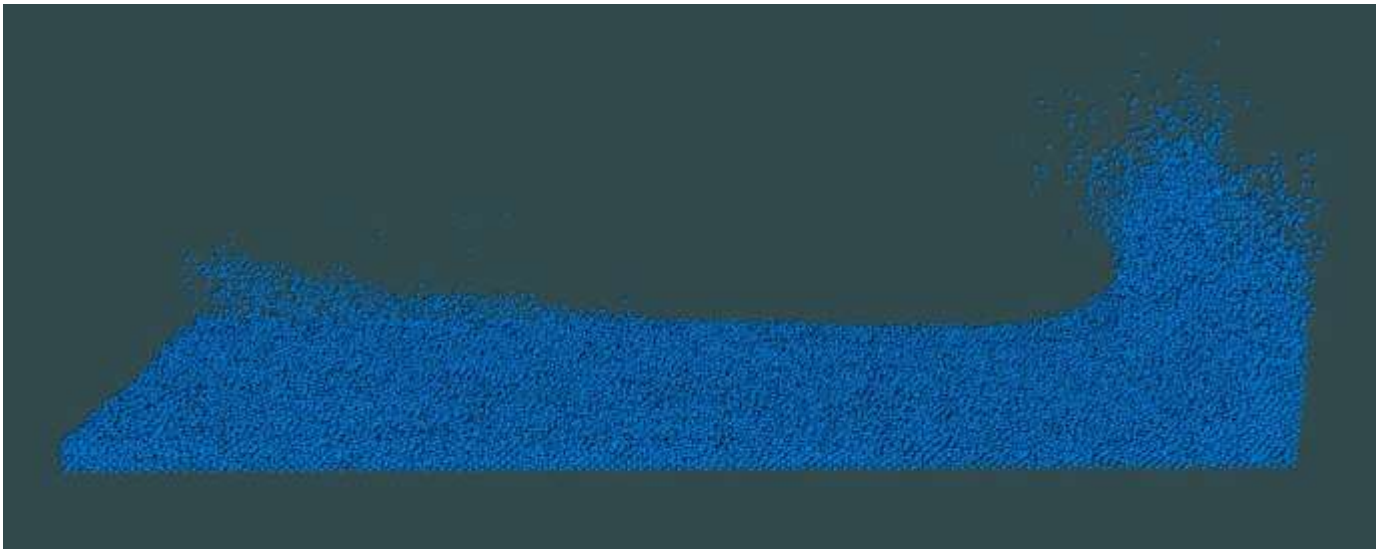


Figure 1 Dam Break 50k particles FPS: 20

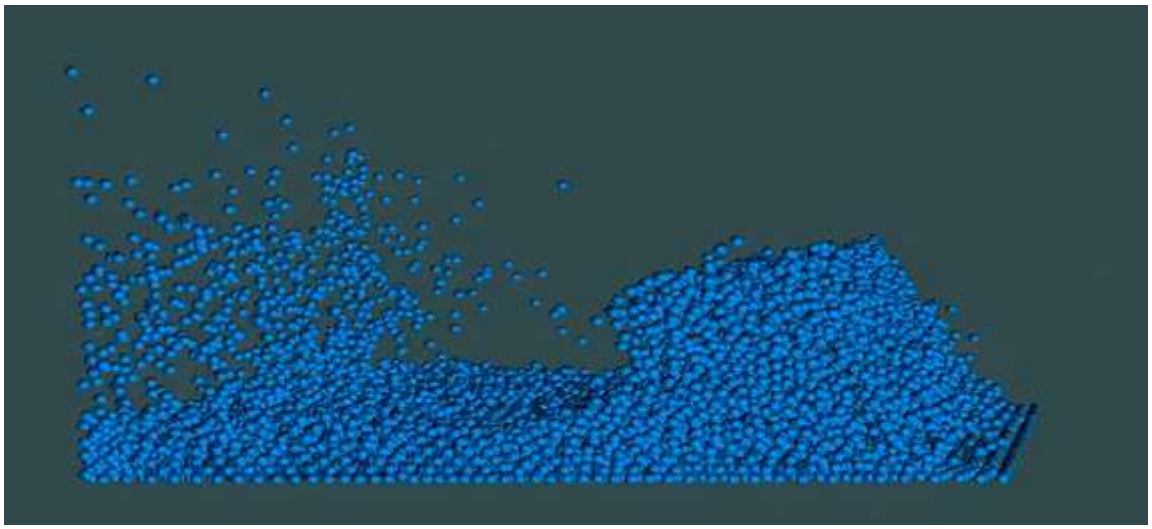


Figure 2 Moving Wall 11.25k particles FPS: 54

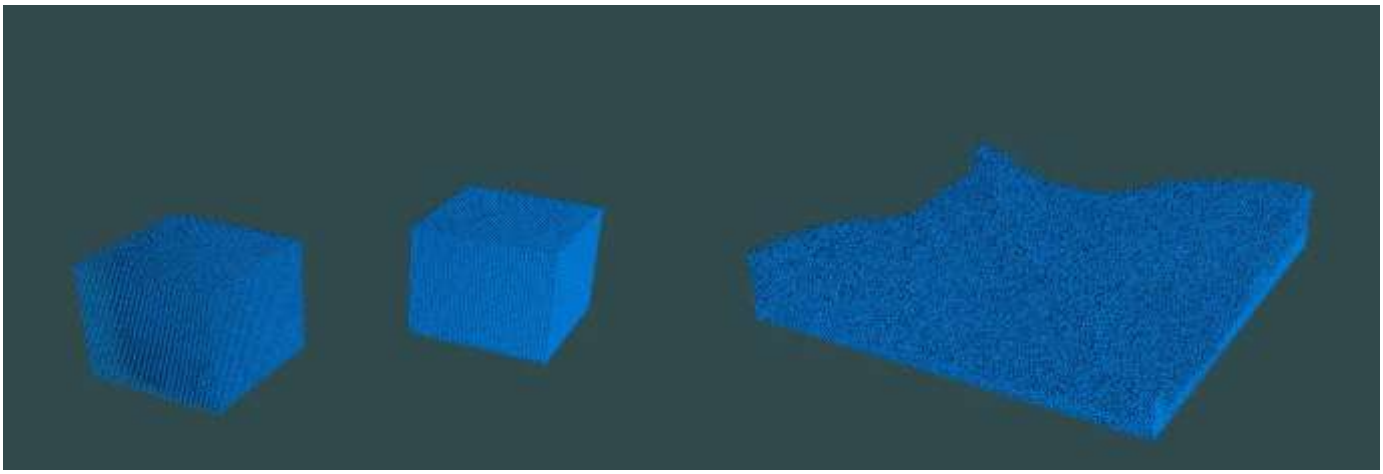


Figure 3 Double Dam Break 128k particles FPS: 9

PBF 的帧数和场景大小、粒子数的多少有关。在简单的 64*64*64 场景中，30k 以下的粒子可以做到 30 帧实时计算。场景可以用键鼠进行 FPS 漫游。

心得体会

这次大程让我体验了从零开始的酸爽感觉。从最开始的粒子系统实现到论文理解、实现，每一步都走的不容易。

最大的收获应当是 CUDA 编程经验，在实现粒子系统的过程中我踩了很多坑：比如.cuh 文件与.h 文件相互包含，CUDA 自定义函数的调用，CUDA 内置函数的调用等。这些都浓缩在了我工程的文件组织上了。

另外就是对粒子系统调参有了初步的了解。粒子系统的调参不仅仅是为了能够达到更好的模拟效果，也是发现系统的不足、漏洞的地方。在调参的过程中，我对初版的程序进行了大量的修改，增加了一些边缘条件的判定，修改了 Demo 的初始模拟参数等。

最后就是，Nvidia Flex 在把 PBF 整合到他们的游戏引擎中去的时候应当还有论文中没有体现的新约束条件，比如下图 Flex 手册中的参数，在论文中并没有被提到。这也应当是网上没有任何一个实现 PBF 能像 Nvidia 一样的原因。

9.1 Parameters

As said, the FlexParams play a big role in the behavior of the particles, especially for fluid simulations. The FlexParams has a set of parameter I hadn't used before for soft bodies since they all relate to fluids.

float	mFluidRestDistance	The distance fluid particles are spaced at the rest density. Range [0, radius]
bool	mFluid	Particles with phase 0 are considered fluid particles and interact using the position based fluids method.
float	mCohesion	Control how strongly particles hold each other together, default: 0.025, Range [0.0, +inf].
float	mViscosity	Smoothes particle velocities using XSPH (Smoothed
float	mFreeSurfaceDrag	Drag force applied to boundary fluid particles.
float	mBuoyancy	Gravity is scaled by this value for fluid particles.

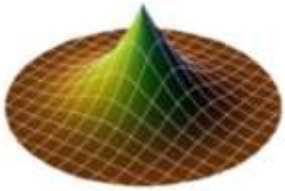
With these parameters in mind, I looked at the demo application and started experimenting with different combinations ranging from viscous behavior to water-like behavior.

另外，为了能够比较效果，我在开发过程中开发了上文提到了两种粒子系统的实现方案。虽然最后采用了在我的实现下，比较快速的第二种。但使用两个系统相互比较的开发过程还是让工作效率提高了不少，毕竟可以知道效果的差距到底出现在哪里，也检查出了一些头脑简单犯下的错误。我想，这也是我一人一队的最大挑战，没有第二个人可以比较、参考，找出不足。

附录

核函数

$$W_{poly6}(\mathbf{r},h)=\frac{315}{64\pi h^9}(h^2-|\mathbf{r}|^2)^3$$



$$\nabla W_{spiky}(\mathbf{r},h)=\frac{45}{\pi h^6}(h-|\mathbf{r}|)^2\frac{\mathbf{r}}{|\mathbf{r}|}$$

参考目录

MACKLIN M., MÜLLER M.: Position based fluids. ACM Trans. Graph. 32, 4 (July 2013), 104:1–104:12. 5

Simon Green, Particle Simulation using CUDA, May 2010

MÜLLER M., HEIDELBERGER B., HENNIX M., RATCLIFF J.: Position based dynamics. Journal of Visual Communication and Image Representation 18, 2 (2007), 109–118. 2, 22