

计算机图形学大作业：《流体模拟》

1711430 江玥

1711447 诸佳昕

计算机科学与技术

2020 年 1 月 6 日

摘要

实现了基础的流体模拟算法，并且在此基础上利用 MarchingCube 算法添加了表面，利用 OpenGL 实现了摄像机、冯氏光照和天空盒。

目录

1 问题描述	3
2 实验环境及运行说明	3
2.1 实验平台	3
2.2 运行说明	3
2.3 总体架构	3
3 流体模拟原理	4
3.1 Navier-Stokes 方程组	4
3.1.1 动量方程简单推导——内力：压力	4
3.1.2 动量方程简单推导——外力	6
3.1.3 动量方程简单推导	6
3.1.4 物质导数	6
3.2 流体算法——SPH 方法	6
4 流体模拟实现	7
4.1 Point	7
4.2 Box	8
4.3 NeighborTable	10
4.4 FluidSystem	11
5 表面生成算法：Marching Cube	14
5.1 Marching Cube 算法原理	14
5.2 Marching Cube 算法实现	16
6 OpenGL 相关	16
6.1 着色器类	16
6.2 摄像机	17
6.3 光照	19
6.4 天空盒	20
7 运行结果	21
8 其他问题记录	23

1 问题描述

- C: 实现一个网格流体模拟器
 - 基础分: 50% - 1) 能够在适当的参数范围内成功进行简单场景的流体模拟 2) 能够提供模拟结果的直观演示 3) 除数学工具函数外不得使用已有的库提供的代码
 - 升级分: 每项升级获得额外10%, 如: 场景中包含运动的固体; 提高固体边界计算的精确度等

图 1: 作业要求

2 实验环境及运行说明

2.1 实验平台

实验平台是 macOS Mojave 10.14.4, 处理器为 2.2 GHz Intel Core i7, 内存为 16 GB 1600 MHz DDR3。

开发语言是 c++, IDE 为 Xcode Version 11.3 (11C29)。

2.2 运行说明

在 source 文件夹下已经给出了需要的 lib 和 include 文件, 并且在 xcode 中已经对相应的 Header Search Paths 和 Library Search Paths 进行了配置。

初始学习 OpenGL 环境配置的过程也记录在了这篇博客中:<http://leflacon.github.io/81586dd4> (当时刚开始用, 因此配置路径十分混乱, 此处对于环境配置的细节不再展开)。

2.3 总体架构

FluidSimulation 文件夹架构如下:

```
.  
SPH_T # project file  
Point.h/cpp # 粒子类  
box.h/cpp # 网格类  
NeighborTable.h/cpp # 邻接表类  
MarchingCube.h/cpp # 表面相关实现  
FluidSystem.h/cpp # 流体系统类
```

```

stb_image.h/cpp # 图像加载库（为了加载图片导入的C++公共
库）
shader.h/cpp # shader 相关函数
Camera.hpp # 摄像机实现
main.cpp # 主函数
*.vs # 顶点着色器
*.frag # 片元着色器
*.tga # 天空盒纹理图片
SPH_T.xcodeproj # Xcode project
source # 编译需要的库文件
fluidsimulation_report.pdf # 报告
README.md # 使用说明
*.mp4 # 录频文件
流体1.0 # 最基础的流体模拟实现代码（一开始未结合OpenGL）

```

3 流体模拟原理

3.1 Navier-Stokes 方程组

$$\frac{\partial u}{\partial t} + u * \nabla u + \frac{1}{\rho} * \nabla p = g + \nu \nabla * \nabla u \quad (1)$$

$$\nabla u = 0$$

第一个方程也叫动量方程，其本质是牛顿第二定律 $F = ma$ 。 \vec{u} 表示速度场中的速度，在三维空间中的分解形式为 (u, v, w) 。 ρ 表示流体密度。 p 表示压力。 g 表示重力加速度。 ν 表示运动粘度（描述了流体在运动时抵抗变形的能力）。

真实的流体，甚至是水这样的液体体积也会改变，但通常变化不大。这样的研究对象称为可压缩流，研究起来很复杂，而宏观上的影响不大，因此初始设定条件为不可压缩流体，对应着 Navier-Stokes 方程组里的第二个式子： $\nabla * u = 0$ 。

3.1.1 动量方程简单推导——内力：压力

内力是影响流体粒子与附近其他粒子相互作用的两种流体力。一种是压力，一种是由流体粘度引起的力。

压力来源：当外力施加在流体上的时候，外力并没有马上传导到整个流体，而是在受力区附近产生高压，高压区推向低压区。压力计算： $-V\nabla p$ 。 $-\nabla p$ 为负压力梯度，因为高压向低压推动，所以取负。体积 V（研究的是不可压缩流体，所以 V 为常量）。

此处再补充一张“参考文献 [3]”中的粒子间的相互作用力示意图，粒子之间在分离的时候产生吸引力，集中的时候产生互斥力。

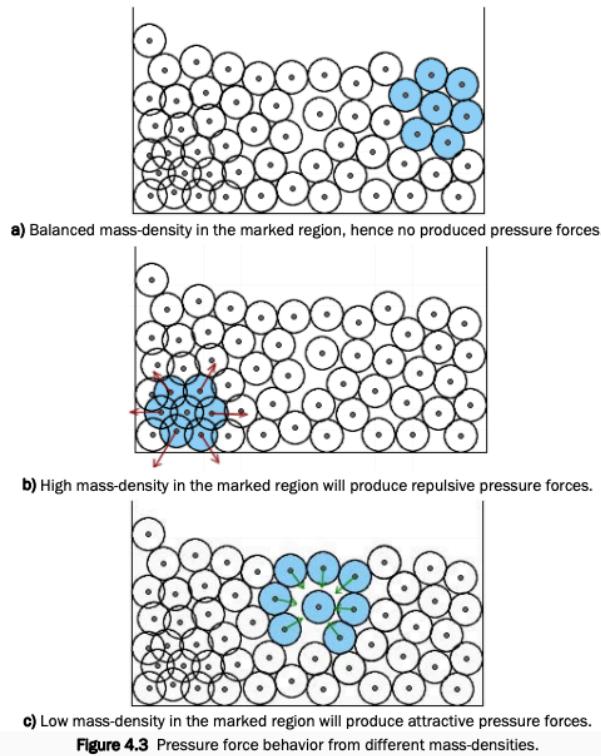


图 2: 粒子间作用力

粘性流体试图抵抗变形，这种内力试图使这个粒子以周围粒子的平均速度运动，即邻近粒子之间的速度差异最小化。在进行空间离散的时候设置的步长是一个有限小的量，不是无限小，这引起了流场变量在这个小局部内的平均，即数值粘性造成了动量的扩散，也可以称为人工粘性/假扩散，如下图：

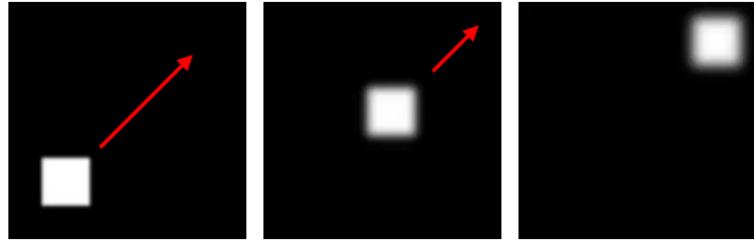


图 3: 随着移动边界模糊

粘度计算: $V\mu\nabla \cdot \nabla u$ 。 μ 为粘滞系数。

3.1.2 动量方程简单推导——外力

外力就是均匀的作用于整个流体的力。主要就是考虑重力: mg 。

3.1.3 动量方程简单推导

把上面这些作用力代入 $F = ma$ 中, 可得:

$$\rho\vec{a} = \rho\vec{g} - \nabla p + \mu\nabla^2\vec{u} \quad (2)$$

进一步化简得:

$$\vec{a} = \vec{g} - \frac{\nabla p}{\rho} + \frac{\mu\nabla^2\vec{u}}{\rho} \quad (3)$$

得到 Navier-Stokes 方程的一个简单形式。

3.1.4 物质导数

q 表示通过流体传输的模拟量之一 (比如温度), 根据链式法则得到下式:

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + \vec{u} * \nabla q \quad (4)$$

$\frac{\partial q}{\partial t}$ 表示了 q 在固定点变化的速度, $\vec{u} * \nabla q$ 纠正多少变化是由于流体流过的差异。

3.2 流体算法——SPH 方法

在拉格朗日视角下的方法。把流体模拟成很多离散的粒子, 只要能描述任意一个粒子的运动状况 (位置随时间变化的规律等), 那么就可以获得整个流体的运动状况。

求解 Navier-Stokes 方程组，引入 SPH 方法（光滑粒子流体动力学方法）。

引入一个光滑核的概念，每个粒子属性都会扩散到它周围，影响周围的粒子，越远那么影响就越小，这种随着距离衰减的函数就是光滑核函数。

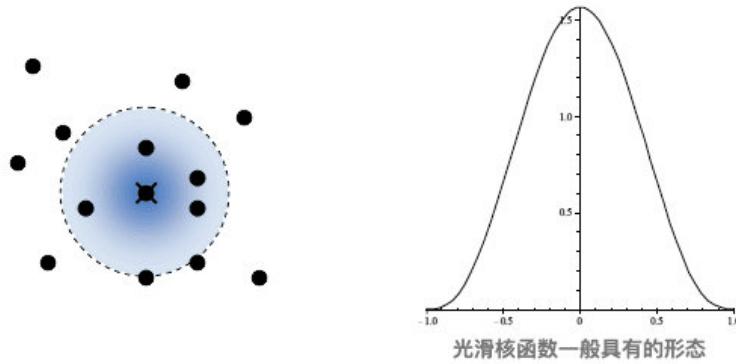


图 4: 光滑核

考虑流体中某点 \vec{r} ，以这个点为圆心，光滑核半径 h 为半径的圆中有其他粒子 $\vec{r}_0, \vec{r}_1, \dots, \vec{r}_j$ ，那么点 \vec{r} 处就有一个关于属性 A 的累加公式：

$$A(\vec{r}) = \sum_j A_j \frac{m_j}{\rho_j} W(\vec{r} - \vec{r}_j, h) \quad (5)$$

其中 A_j, m_j, ρ_j 代表其他粒子的属性 A/质量/密度， W 是光滑核函数。可见这个方法对于光滑核函数选取的要求很高。

再考虑流体中一个位置为 \vec{r}_i 的点，由牛二推导的式子可得：

$$\vec{a}(\vec{r}_i) = \vec{g} - \frac{\nabla p(\vec{r}_i)}{\rho(\vec{r}_i)} + \frac{\mu \nabla^2 \vec{u}(\vec{r}_i)}{\rho(\vec{r}_i)} \quad (6)$$

其中 $u(\vec{r}_i), p(\vec{r}_i), \rho(\vec{r}_i)$ 分别代表此处的速度/压力/密度。

接下来就是每一项分开求解的过程，把关于属性 A 的累加公式里的 A 换成要求解的属性，取适当的光滑核函数，经过相应推导得到该属性计算公式。

4 流体模拟实现

4.1 Point

首先建立一个粒子结构体。

值得一提的是，这里的密度场是一个标量场，这在后续 MarchingCube 的实现中有重要作用。

```
1 //  
2 struct Point{  
3     glm::vec3 pos; // (x, y, z)  
4     float density; //  
5     float pressure; //  
6     glm::vec3 acceleration; //  
7     glm::vec3 velocity; //  
8     glm::vec3 final_velocity; //  
9     int next; //  
10};
```

然后是一个粒子缓存类用来保存粒子。

```
1 //  
2 class PointBuffer{  
3 public:  
4     PointBuffer();  
5     virtual ~PointBuffer();  
6     void reset(unsigned int capacity); //  
7     unsigned int size() const {return pointbuffer_cnt;} //  
8  
8     Point* get(unsigned int index) const {return pointbuffer_p+  
9         index;} // index  
9     Point* addPointReuse(); //  
10 private:  
11     Point* pointbuffer_p; //  
12     unsigned int pointbuffer_cnt; //  
13     unsigned int pointbuffer_capacity; //  
14};
```

4.2 Box

后续表示三维空间的时候也用“min 左下 max 右上”这样的表示方式。

```
1 //  
2 class FluidBox{  
3 public:  
4     FluidBox();  
5     FluidBox(glm::vec3 aMin,glm::vec3 aMax);  
6     // min      max  
7     glm::vec3 min;  
8     glm::vec3 max;  
9 };
```

然后要对空间网格进行划分。

依据 SPH 算法的原理，流体粒子和粒子间相互影响，我们要计算的就是一个粒子周围的粒子对它的影响。那么从时间复杂度考虑，遍历一遍所有粒子来计算相邻粒子是哪些就是 $O(n^2)$ 的。如果引入网格，然后将粒子保存在网格中，那么每次寻找邻域粒子的时候只需要查找周围网格内的粒子即可。

此外，在学习过程中发现也有 SPH 无网格法，并且也有人实现了大规模计算。时间有限并未了解其原理，此处仅记录不展开说明。

```
1 //  
2 class FluidGrid{  
3 private:  
4     //  
5     std::vector<int> grid_points;//  
6     glm::vec3 grid_min;//  
7     glm::vec3 grid_max;//  
8     glm::ivec3 grid_standards;//          x*y*z  
9     glm::vec3 grid_size;//  
10    glm::vec3 grid_offset;//  
11 public:  
12    FluidGrid(){}  
13    ~FluidGrid(){}  
14    const glm::ivec3* getGridRes() const{return &grid_standards  
15        ;}  
16    const glm::vec3* getGridMin() const{return &grid_min;}
```

```

16     const glm::vec3* getGridMax() const{return &grid_max;}
17     const glm::vec3* getGridSize() const{return &grid_size;}
18     float getOffset(){return grid_offset.x;}
19     void init(const FluidBox& box,float sim_scale,float
20             cell_size,float border,int* rex);//
21     int getGridPoints(int gridIndex);//
22     int getGridIndex(float px,float py,float pz);//(px,
23             py,pz)
24     void insertPoints(PointBuffer*pointBuffer);//

25     void getNeighborGrids(const glm::vec3& p,float radius,int *
26             gridCell);//
27 };

```

4.3 NeighborTable

最后一个基础结构类是邻接表类。

```

1 class NeighborTable{
2 private:
3     union PointExtraData{
4         struct{
5             unsigned extra_neighbor_offset:24;// 
6             unsigned extra_neighbor_cnt:8;// 
7         };
8         unsigned int neighborData;// 
9     };
10    PointExtraData* point_extra_data;// 
11    unsigned int neighbor_point_cnt;// 
12    unsigned int neighbor_point_capacity;// 
13    unsigned char* neighbor_point_buffer;// 
14    unsigned int neighbor_buffer_size;// 
15    unsigned int neighbor_buffer_offset;// 
16    unsigned short cur_point;// 

```

```

17     int cur_neighbor_cnt; //
18     unsigned short cur_neighbor_index[MAX_NEIGHBOR_CNT]; //

19     float cur_neighbor_distance[MAX_NEIGHBOR_CNT]; //

20 public:
21     NeighborTable();
22     ~NeighborTable();
23     void reset(unsigned short pointCounts); //
24     void pointInitCur(unsigned short point_index); //

25     bool addNeighbor(unsigned short point_index, float distance)
26         ; //
27     void commitToTable(); //
28     void updatePointBuffer(unsigned int need_size); //

29     int getNeighborCnt(unsigned short point_index); //

30     void getNeighborInfo(unsigned short point_index, int index,
31             unsigned short& neighborIndex, float& neighborDistance);
32             //           point_index           index
33 };

```

4.4 FluidSystem

最后是流体系统类的基础实现。

根据 OpenGL 循环渲染的原理，编写一个 tick() 函数设置好时间间隔用于更新粒子的位置，那么在循环渲染的时候调用这个函数，就可以实现流体粒子的位置变化。

```

1 class FluidSystem{
2 private:
3     //
4     PointBuffer fluid_point_buffer;
5     FluidGrid fluid_grid;

```

```
6     NeighborTable fluid_neighbor_table;
7     std::vector<float>fluid_point_pos; // 
8     //SPH
9     float kernel_poly6;
10    float kernel_spiky;
11    float kernel_viscosity;
12    int m_rexSize[3]; //
13    float fluid_point_radius; //
14    float fluid_scale; //
15    float fluid_viscosity; //
16    float fluid_density; //
17    float fluid_point_mass; //
18    float smooth_radius; //
19    float gas_const_k; // k
20    float boundary_stiffness; //
21    float boundary_dampening; //
22    float fluid_limit_speed; //
23    glm::vec3 vec_gravity; //
24    FluidBox fluid_wall_box; //
25    void initFluidSystem(unsigned short maxPointCounts, const
26                          FluidBox& wallBox, const FluidBox& initFluidBox, const glm
27                          ::vec3& gravity); //
28    void computerPressure(); //
29    void computerAcceleration(); //
30    void computerPos(); //
31    void createFluidPoint(const FluidBox& fluidBox, float
32                          spacing); //
33 public:
34     FluidSystem();
35     void init(unsigned short maxPointCounts,
36               const glm::vec3& wallBox_min, const glm::vec3&
37               wallBox_max,
38               const glm::vec3& initFluidBox_min, const glm::vec3
```

```

35         & initFluidBox_max ,
36         const glm::vec3& gravity) {
37     initFluidSystem(maxPointCounts, FluidBox(wallBox_min ,
38             wallBox_max),
39             FluidBox(initFluidBox_min, initFluidBox_max),
40             gravity);
41 }
42     unsigned int getPointSize() const; //      size
43     unsigned int getPointCnt() const; // 
44     const Point* getPointBuf() const; // 
45     void tick(); // 
46     ~FluidSystem();
47 };

```

此处还需要展开说明的是几个计算函数的推导，公式推导主要参考了参考文献 [5]。

基于上文原理部分推导出的公式，继续引入光滑核函数，光滑核函数的选取也十分重要，它对于流体模拟的速度、稳定性以及质量都有很大的影响。在参考文献 [4] 中提出了一个 Poly6 函数，这是目前大多数 SPH 算法都使用的函数，形式如下：

$$W_{Poly6} (\vec{r}, h) = \begin{cases} K_{Poly6} (h^2 - r^2)^3 & , 0 \leq r \leq h \\ & \text{with } r = |\vec{r}| \\ 0 & , \text{otherwise} \end{cases} \quad (7)$$

但是 Poly6 核的梯度在中心变为 0，这样对压力计算是不好的，因为粒子十分接近时应该有很大的压力（互斥），所以就需要一个梯度在中心有较大取值的平滑核来插值压力，也就是 Spiky 核，形式如下：

$$W_{Spiky} (\vec{r}, h) = \begin{cases} K_{Spiky} (h - r)^3 & , 0 \leq r \leq h \\ & \text{with } r = |\vec{r}| \\ 0 & , \text{otherwise} \end{cases} \quad (8)$$

最后得到的密度计算公式为：

$$\rho(r_i) = m \frac{315}{64\pi h^9} \sum_j (h^2 - |\vec{r}_i - \vec{r}_j|^2)^3 \quad (9)$$

结合压力和粘度对加速度的影响得到的加速度的计算公式为：

$$\vec{a}(r_i) = \vec{g} + m \frac{45}{\pi h^6} \sum_j \left(\frac{p_i + p_j}{2\rho_i \rho_j} (h - r)^2 \frac{\vec{r}_i - \vec{r}_j}{r} \right) + m\mu \frac{45}{\pi h^6} \sum_j \frac{\vec{u}_j - \vec{u}_i}{\rho_i \rho_j} (h - r) \text{ with } r = |\vec{r}_i - \vec{r}_j| \quad (10)$$

至此，流体模拟的所有基础实现已经完成。

5 表面生成算法：Marching Cube

然后引入表面生成算法：Marching Cube。

5.1 Marching Cube 算法原理

这是一个高分辨率 3D 表面构造算法。这个算法最初用在医学上对 3D 解剖结构进行可视化。

首先是“体元（Cell）”的概念。使用分治法将表面定位在由八个像素创建的逻辑立方体中，两个相邻的切片各四个。如下图：

体元就是 8 个体素构成的方格，每个体素都被 8 个体元共享。

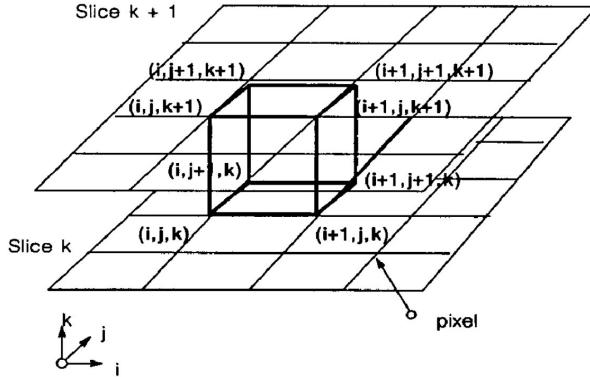


Figure 2. Marching Cube.

图 5: 体元

这个算法是基于标量场的，所以本来是不知道这个网格点在表面内还是表面外的，需要借助一个隐函数， $f(x, y, z) = 0$ 表示表面上的点集。

如果这 8 个顶点连续的两个有一个 > 0 一个 < 0 ，也就是一个在表面外一个在内，那么表面肯定在这两个顶点之间，那么之后通过插值就可以获得表面的位置。

每个立方体有八个顶点，因此曲面与立方体相交的方式有 $2^8 = 256$ 种。

然后对着 256 种方式考虑两种对称性：

1. 第一种是互补的情况，因为取 x 个点其实和取互补的 $8 - x$ 个是一样的表面，所以只需要考虑 0-4 个点的情况，那么总方式减为 $256/2 = 128$ 种
2. 第二种是旋转的情况，在上面只考虑 0-4 个点的基础上加上旋转，最后可以总结出下图 15 种类型的结构：

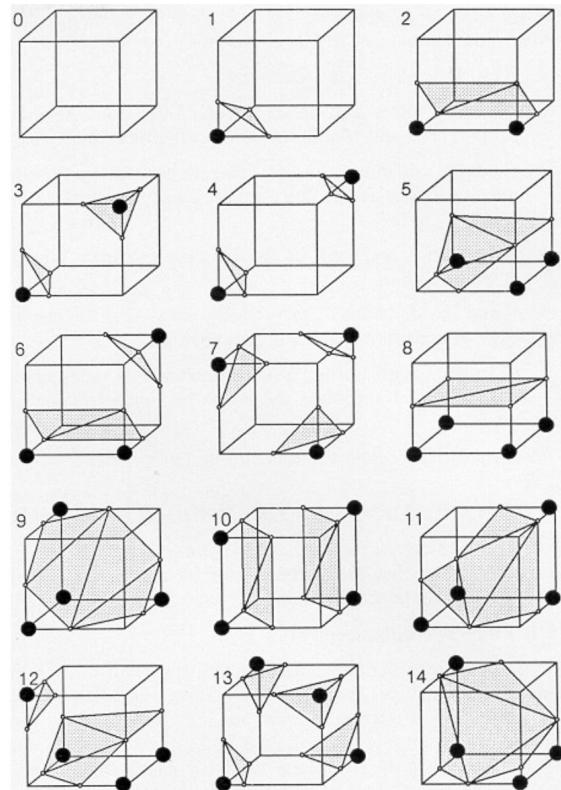


Figure 3. Triangulated Cubes.

图 6: 15 种结构

接下来根据顶点的状态为每种情况创建一个索引。使用下图中的顶点编号，八位索引的每个位表示一个顶点 v 。

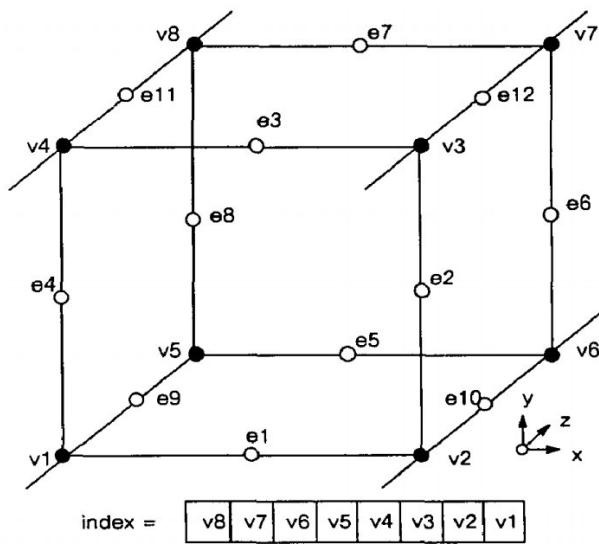


Figure 4. Cube Numbering.

图 7: 索引

最后为每个三角形顶点计算一个单位法线。渲染算法使用该法线生成 Gouraud-shaded 图像。

5.2 Marching Cube 算法实现

对应的实现在 Marching Cube.h/cpp 中。在 SPH 算法的实现过程中已经建立了空间网格，因此 Marching Cube 算法中无需另外构造网格。

6 OpenGL 相关

以上算法部分就结束了，接下来简略介绍 OpenGL 相关的部分，这部分内容实在是太过繁杂，笔者水平有限，因此以下不详细展开，代码中补充了不少教程内的注释。

6.1 着色器类

Shader.h/cpp 实现了封装了一些着色器类的基本函数，按照编译定点着色器，编译片元着色器，链接着色器程序的流程实现着色器的使用。

6.2 摄像机

摄像机的主要实现在 Camera.hpp 文件中，原理是通过把场景中的所有物体往相反方向移动的方式来模拟出摄像机，产生一种我们在移动的感觉，而不是场景在移动。

```
1 enum Camera_Movement{
2     FORWARD,
3     BACKWARD,
4     LEFT,
5     RIGHT
6 };
7 const float YAW=0,PITCH=0,SPEED=2.5,SENSITIVITY=0.1,ZOOM=45;
8 //                                     /          (Camera/View Space)
9 //
```



```
10 class Camera{
11 public:
12     glm::vec3 Position,Front,Right,WorldUp;
13     float Yaw,Pitch,MovementSpeed,MouseSensitivity,Zoom;
14     Camera(glm::vec3 position=glm::vec3(0.0,20.0f,50.0f),glm::
15         vec3 up=glm::vec3(0.0f,1.0f,0.0f),float yaw=YAW,float
16         pitch=PITCH){
17         Position=position;
18         WorldUp=up;
19         Yaw=yaw;
20         Pitch=pitch;
21         Front=glm::vec3(0.0f,0.0f,1.0f);
22         MouseSensitivity=SENSITIVITY;
23         MovementSpeed=SPEED;
24         Zoom=ZOOM;
25         updateCameraVectors();
26     }
27     Camera(float posX,float posY,float posZ,float upX,float upY
28           ,float upZ,float yaw,float pitch){
```

```
26     Position=glm::vec3(posX,posY,posZ);
27     WorldUp=glm::vec3(upX,upY,upZ);
28     Yaw=yaw;
29     Pitch=pitch;
30     Front=glm::vec3(0.0f,0.0f,1.0f);
31     MouseSensitivity=SENSITIVITY;
32     MovementSpeed=SPEED;
33     Zoom=ZOOM;
34     updateCameraVectors();
35 }
36 //
37 void updateCameraVectors(){
38     glm::vec3 front;
39     front.x=sin(glm::radians(Yaw))*cos(glm::radians(Pitch))
40         ;//
41     front.y=sin(glm::radians(Pitch));
42     front.z=-cos(glm::radians(Yaw))*cos(glm::radians(Pitch))
43         );
44     Front=glm::normalize(-front);
45     Right=glm::normalize(glm::cross(glm::vec3(0.0f,1.0f,0.0
46         f),Front));
47     WorldUp=glm::normalize(glm::cross(Front,Right));
48 }
49 //
50 void processKeyboard(Camera_Movement direction,float
51     delta_time){
52     float velocity=MovementSpeed*delta_time;
53     if(direction==FORWARD)Position+=-Front*velocity;
54     if(direction==BACKWARD)Position=-Front*velocity;
55     if(direction==LEFT)Position-=Right*velocity;
```

```

55         if(direction==RIGHT)Position+=Right*velocity;
56     }
57     void processMouseMovement(float xOffset,float yOffset){
58         //
59         xOffset *= MouseSensitivity;
60         yOffset *= MouseSensitivity;
61         Yaw+=xOffset;
62         Pitch-=yOffset;
63         //
64         //                                     89          90
65         //                                     -89
66
67         if(Pitch>89.0f)Pitch=89.0f;
68         if(Pitch<-89.0f)Pitch=-89.0f;
69         updateCameraVectors();
70     }
71     //                                     (Zoom)
72     void processMouseScroll(float yoffset){
73         if(Zoom>=5.0f&&Zoom<=70.0f)Zoom-=yoffset;
74         if(Zoom<=5.0f)Zoom=5.0f;
75         if(Zoom>=70.0f)Zoom=70.0f;
76     };

```

6.3 光照

冯氏光照，这部分没有详细研究，大多参考了参考文献 [8] 和参考文献 [9] 中光照部分的相关实现。

在流体的片着色器中，调节 allColor 变量里天空盒颜色的参数，加权相加。

```

1 #version 330 core
2 out vec4 FragColor;
3

```

```
4 //  
5 in vec3 Normal;  
6 in vec3 Position;  
7 //  
8 uniform vec3 cameraPos;  
9 uniform samplerCube skybox;  
10  
11 void main(){  
12     float ratio=1.00/1.33;  
13     //  
14     vec3 I=normalize(Position-cameraPos);  
15     vec3 R1=reflect(I,normalize(Normal));  
16     vec3 R2=refract(I,normalize(Normal),ratio);  
17     // vec4 allColor=vec4(1.0,1.0,1.0,1.0);  
18     vec4 allColor=vec4(texture(skybox,R1).rgb*0.4+texture(  
19         skybox,R2).rgb*0.6,1.0);  
20     FragColor=allColor;  
}
```

6.4 天空盒

天空盒的素材来自这个网站: <http://www.custommapmakers.org/skyboxes.php>。

具体的实现在 main.cpp 中。加入纹理素材时图片读取部分引入了一个 C++ 公共库 stb_image.h, 这个库实现了从文件/内存中加载/解码图像: JPG, PNG, TGA, BMP, PSD, GIF, HDR, PIC。在 stb_image.cpp 中通过定义 STB_IMAGE_IMPLEMENTATION, 预处理器会修改头文件, 让其只包含相关的函数定义源码, 等于是将这个头文件变为一个.cpp 文件。

这里值得一提的是下面这段天空盒的顶点着色器程序, 里面有一个非常巧妙的地方: 不是直接将变换之后的坐标传入 gl_Position 中, 在传入前用其 W 分量替换了 Z 分量的值, 变成了 xyww。

这是因为在光栅化阶段中光栅器会对 gl_Position 执行透视除法 (除以其 W 分量), 将 Z 分量替换成 W 分量的值, 则在透视除法之后这个向量的 Z 分量的值就成为了 1.0,

这样这个片元就会始终位于远平面的位置，这意味着在深度测试的过程中，天空盒的片元和场景中任意模型的片元比较都会失败。这样天空盒就只会是填满场景模型留下的背景空隙，也就是被之后要绘制的物体遮挡住，这也是我们希望得到的效果。

```
1 #version 330 core
2 layout (location=0) in vec3 aPos;
3 layout (location=1) in vec3 aNormal;
4 out vec3 TexCoords;
5
6 uniform mat4 projection;
7 uniform mat4 view;
8
9 void main(){
10     TexCoords=aPos;
11     vec4 pos=projection*view*vec4(aPos,1.0);
12     gl_Position=pos.xyww;
13 }
```

7 运行结果

粒子效果如图：

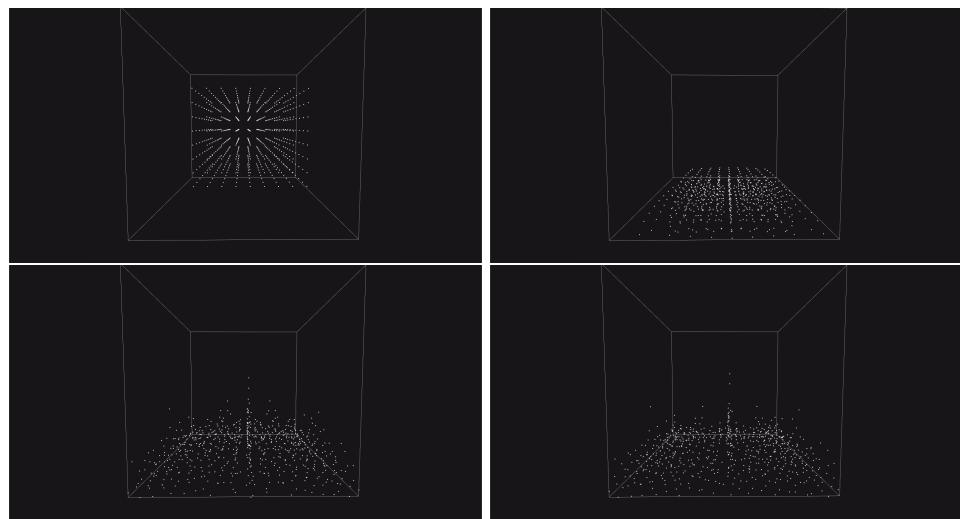


图 8：运行结果组一

流体效果如图：

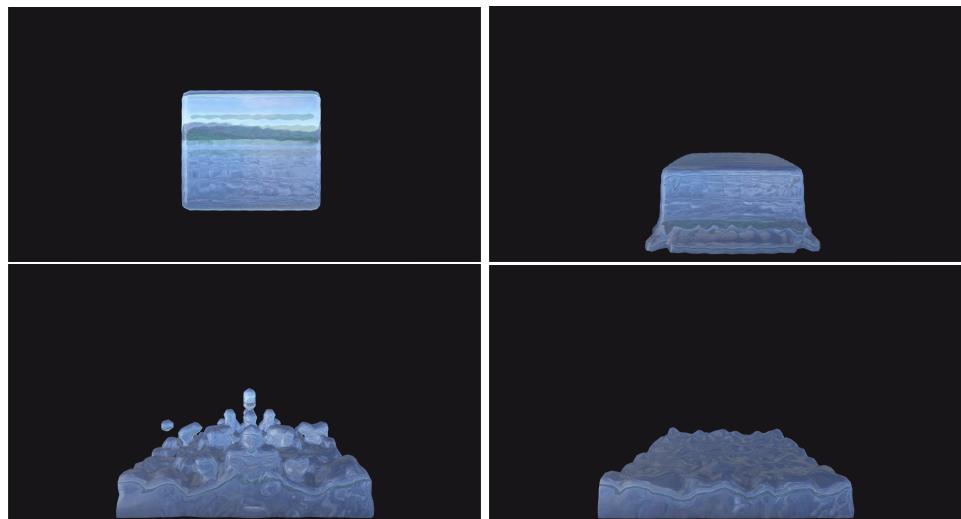


图 9：运行结果组二

加入天空盒后效果如图，最后两张通过鼠标移动改变了 Camera 的视角：

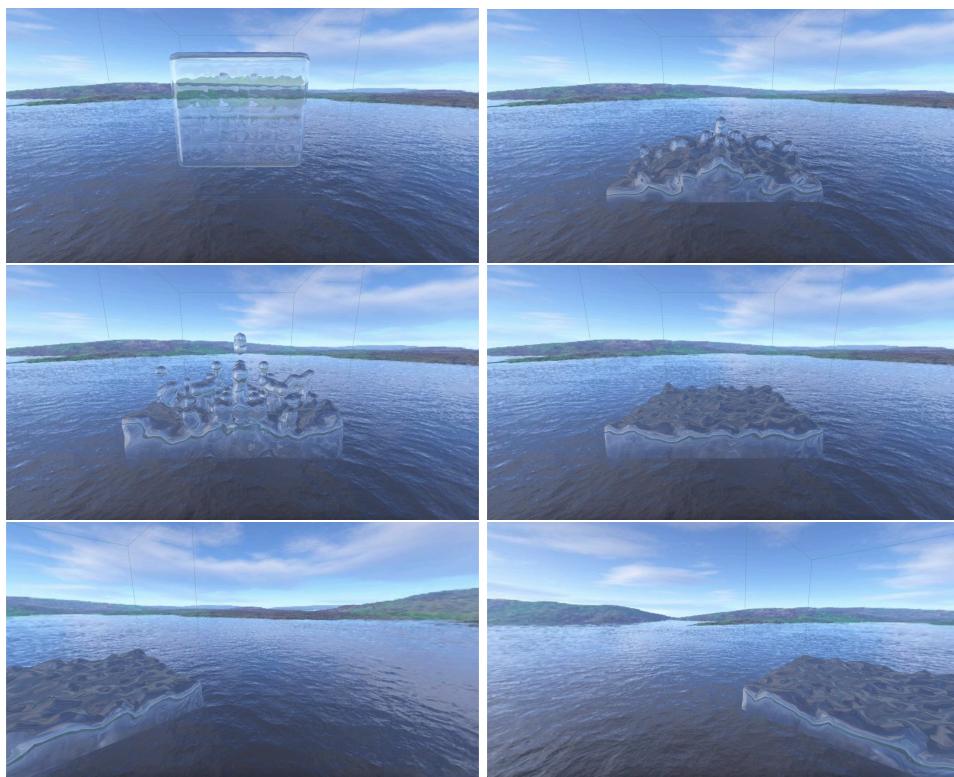


图 10: 运行结果组三

8 其他问题记录

在 Xcode 生成可执行文件的过程中，在本机上可以生成可执行文件并运行，但是换了一台电脑之后无法运行，输出的错误信息如下：

```
dyld: Library not loaded: /Users/mac/Downloads/GLTools-master/
      build/libgltools.dylib
      Referenced from: /Users/mac/Desktop/FluidSimulation/bin/SPH_T
      Reason: image not found
```

尝试了各种方法，包括但不限于生成 framework 时动态库改为静态库/把 framework 的状态由强引用改为弱应用/通过 New copy Files Phase 的方式把 framework 直接拷贝到项目中等，但是都以失败告终。

笔者开发经验不足，对于环境配置方面的理解还很浅薄，此问题最终还是没有解决。

参考文献

- [1] nteractiveComputerGraphics/SPH-Tutorial[OL]Github,2019-06-01
- [2] Clavet, Simon, Philippe Beaudoin, and Pierre Poulin.Particle-based viscoelastic fluid simulation[C]ACM,2005
- [3] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean.Efficient Estimation of Word Representations in Vector Space[C]ICLR,2013-12-07
- [4] Müller, Matthias, David Charypar, and Markus Gross.Particle-based fluid simulation for interactive applications[C]ACM Eurographics Association,2003
- [5] 靳超.SPH 算法简介 [OL]thecodeway.com,2014
- [6] cerrno/pcisph[OL]Github,2018-12-30
- [7] 硕鼠酱. 电影工业中的流体模拟 [OL] 知乎,2017-05-19
- [8] 0 小龙虾 0.SHP (光滑粒子流体动力学) 流体模拟实现 [OL]CSDN,2019
- [9] LearnOpenGL-CN.LearnOpenGL[OL]Github,2019-11-26
- [10] William E. Lorensen,Harvey E. Cline.Marching cubes: A high resolution 3D surface construction algorithm[C]SIGGRAPH,1987
- [11] Sean Barrett.nothings/stb[OL]Github,2019-10-29
- [12] 第二十五课天空盒-现代 OpenGL 教程 [OL] 极客学院 Wiki,2018-11-28
- [13] dr.R.T.Tan.Fluid Simulation for Computer Graphics[D]2013
- [14] 谭捷. 基于物理的流体动画研究 [D] 上海交通大学,2009

最后，真诚的感谢所有直接或间接提供帮助的人！