南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机图形学期末实验报告

A1——光线跟踪核心算法改进
A2——流体模拟器算法改进

姓名：袁鑫明 高路博

学号：2113020 2111172

年级：2021 级

专业：计算机科学与技术

指导教师：任博

2024 年 6 月 15 日

目录

一、 概述

本次大作业我们同时选做了框架一和框架二。

框架一：实现了 Whitted-Style 递归光线追踪算法（基础要求），除此之外尝试使用俄罗斯轮盘赌进行优化，优化效果最高达到 28.8%，同时尝试新增了 Dielectric 绝缘体材质来让算法体现得更加完整（材质效果不佳）。

框架二：完成了在拉格朗日法和欧拉法下的 2D（基础要求）和 3D 流体模拟，并尝试进行了并行加速（这个效果不佳）。

注：本次大作业编写过程中使用了 ChatGPT：

- 用于生成原始框架（包括渲染和流体框架）里面给定的函数的注释，目的在于快速理解框架，并为后续实验更好的调用框架已有的函数，同时，由于 Github Copilot 的存在，在编写代码时，在代码后面加上“//”就会自动生成注释，所以我们在代码中添加了非常多的注释来帮助我们理解和实现代码，包括我们亲自实现的代码也加了较多的注释。
- 在渲染框架中的 Dielectric 材质实现的代码中，针对折射部分的 refract 函数（以及后续 shade 的相关调用）使用了 ChatGPT，使用方法是将写好的函数定义发给它，然后让它生成折射公式的代码实现。
- 在撰写报告时，Whitted-Style 算法介绍中的公式部分使用了 ChatGPT 协助编写，使用方法为“将算法的公式转为 latex”

关键字：Whitted-style 递归光线追踪算法，Dielectric 绝缘体材质，俄罗斯轮盘赌、拉格朗日流体模拟、欧拉流体模拟

二、 框架一

光线追踪的核心算法大致分为光线投射、经典光线追踪、递归式光线追踪和路径追踪等几类，框架中已有的算法是光线投射和简单路径追踪，我们在阅读并熟悉已有框架的基础上，实现了 Whitted-Style 递归光线追踪算法。

(一) Whitted-Style 递归光线追踪

在广泛阅读光线追踪资料以及听 Games101 课程之后，我们得知光线追踪本质上就是从摄像机出发，往每一个像素点投光线，投到已有的场景中，找到与物体的交点，然后和光源连线，是否在阴影中以及交点的颜色亮度等，最终确定该像素的结果。

而 Whitted-style 递归光线追踪则是传统的光线追踪的改进版本，因为实际生活中光线又不是投射到一点之后就停止了，光线还会有反射，遇到透明物体会有折射，本算法递归地计算光线的反射和折射路径，就可以模拟出更复杂的光学现象，也更接近现实生活，更具备真实性，如镜面反射、透明物体的折射和阴影等。

1. 递归光线追踪的公式

根据提出递归式光线追踪的 Whitted 在论文中所述，他提出的光线追踪公式如下：

$$I = I_a + k_d \sum_{j=1}^{N} (N \cdot L_j) + k_s S + k_t T$$

其中：

- I 是最终的光照强度。
- I_a 是环境光强度。
- k_d 是漫反射系数。
- N 是表面法向量。
- L_j 是从表面点到第 j 个光源的方向向量。
- k_s 是镜面反射系数。
- S 是反射光线的强度，通过递归计算得到。
- k_t 是折射系数。
- T 是折射光线的强度，通过递归计算得到。

2. 实现的算法步骤

Whitted-style 递归光线追踪算法的伪代码结构如下：

```
RayTracing(original_point, ray_direction, objects, depth)
{
    if (depth > maxDepth)
        return color(0,0,0);

    if (IsHitObject(original_point, ray_direction, objects))
    {
        hitPoint = GetHitPoint();
        normal = GetNormal();

        reflectionDirection = reflect(ray_direction, normal);
        refractionDirection = refract(ray_direction, normal); //如果没有折射，舍弃折射项

        local_color = BlinnPhongShader(original_point, normal, light_position)

        return local_color
             + k1 * RayTracing(hitPoint, reflectionDirection, objects, depth + 1)
             + k2 * RayTracing(hitPoint, refractionDirection, objects, depth + 1);
    }
    else
        return background_color;
}
```

图 1: 摘自网络

正如论文中的公式所言，其实就是原本颜色加上反射颜色，然后有折射的话再加上折射颜色。在参考伪代码的基础上，我们结合不同的材质写出相应的代码，并结合我们自己的理解做了一些额外工作，但整体流程基本上是一致的：

- 首先限制递归次数、衰减系数、判断场景光源等；
- 判断发出光线是否击中物体，如果击中物体，则判断物体是否遮挡；
- 在不遮挡的情况下，根据不同材质递归计算反射或折射的颜色；
- 最终把反射或折射或二者与原先颜色相加，得到最终的颜色。

递归光线追踪核心就是对击中之后生成的反射和折射光线都再进行递归的光线追踪计算来实现一步步地重复，这个重复可以是有限的，比如我们设置了最大递归深度为 50，在达到这个设定次数之后就让算法停止了。

算法的核心代码如下（完整代码见源文件）：

```
// 如果击中了物体
if (closestHitObj) {
    // 获取击中的物体的信息
    auto& hitRec = *closestHitObj;// 找到击中物体
    auto& hitMaterial = hitRec.material;// 获取击中物体的材质
    auto& hitNormal = hitRec.normal;// 获取击中物体的法向量
    auto& hitPoint = hitRec.hitPoint;// 获取击中物体的交点
    auto out = glm::normalize(l.position - hitPoint);// 计算出射光线的方向
    // 如果出射光线与法向量的点积小于 0，说明光线与法向量的夹角大于 90 度
    // 即光源在物体的背面，所以返回 rgb000--黑色
    if (glm::dot(out, hitNormal) < 0) {
        return {0, 0, 0};
    }
    auto distance = glm::length(l.position - hitPoint);// 计算光源到交点的距离
    auto shadowRay = Ray{hitPoint, out};// 创建阴影光线
    auto shadowHit = closestHit(shadowRay);// 找到阴影光线击中的最近的物体
    // 计算着色，c 是光照强度
    auto c = shaderPrograms[hitMaterial.index()]->shade(-r.direction, out, hitNormal);
    // 根据不同材质分成不同的情况
    auto& MaterialForSwitch = scene.materials[hitMaterial.index()];
    int TypeForSwitch = typeofMaterial(MaterialForSwitch);
    if(randomFloat > stopP){// 俄罗斯轮盘赌，大于才递归，小于 stopP 则直接终止递归
        switch(TypeForSwitch){
            case 0:{// Lambertian 材质，只有漫反射，保持原来的就行，不用递归
                // 如果没有阴影，或者有阴影但是阴影的距离大于光源到交点的距离，返回光照强度
                if ((!shadowHit) || (shadowHit && shadowHit->t > distance)) {
                    return c * l.intensity;// 没有遮挡，返回光照强度
                }
                else {
                    return Vec3{0};// 有遮挡，返回 rgb000--黑色
                }
                break;
            }
            case 1:{// Phong 材质
                RGB plusRefLect = Vec3{0};// 先默认为黑色
                // 新增反射光线，反射方向 = 入射方向 -2*(入射方向 * 法向量)* 法向量
                // 计算反射光线方向
                // 必须是入射光线相对于表面法线的反射方向！不是光源方向！
                // 直接用现成的 reflect 函数计算反射光线
                Vec3 reflectDir = reflect(glm::normalize(r.direction), hitNormal);
                Ray reflectRay{hitPoint, reflectDir}; // 反射光线
                //Ray reflectRay{hitPoint, reflect(hitNormal, out)};// 反射光线 好像不太对
                // 新增反射衰减
```

```
44                  float reflectDecay = attenuation * 0.8f;
45                  // 递归计算反射光线的颜色
46                  auto reflectColor = trace(reflectRay, recursionDepth + 1, reflectDecay);
47                  // 如果没有阴影，或者有阴影但是阴影的距离大于光源到交点的距离，返回光照强度
48                  if ((!shadowHit) || (shadowHit && shadowHit->t > distance)) {
49                      plusRefLect = c * l.intensity
50                                  + reflectColor;
51                      return plusRefLect;// 没有遮挡，返回光照强度
52                  }
53                  else {
54                      RGB plusRefLect = Vec3{0}
55                                      + reflectColor;
56                      return plusRefLect;// 有遮挡，返回黑色
57                  }
58                  break;
59              }
60              case 2:{// Dielectric 材质如玻璃，这个有折射
61                  RGB plusRefLectRefrect = Vec3{0};// 先默认为黑色
62                  float refract_index = 1.5f;// 先用 1.5 代替吧
63                  // 反射 和上面一样 用 reflect 函数计算反射光线
64                  Vec3 reflectDir = reflect(glm::normalize(r.direction), hitNormal);
65                  Ray reflectRay{hitPoint, reflectDir}; // 反射光线
66                  float reflectDecay = attenuation * 0.8f; // 新增反射衰减
67                  // 递归计算反射光线
68                  auto reflectColor = trace(reflectRay, recursionDepth + 1, reflectDecay);
69                  // 折射
70                  Vec3 refractDir = refract(hitNormal,
71                  glm::normalize(r.direction), refract_index);// 先用 1.5 代替吧
72                  Ray refractRay{hitPoint, refractDir}; // 折射光线
73                  float refractDecay = attenuation * 0.7f; // 新增折射衰减
74                  // 递归计算折射光线
75                  auto refractColor = trace(refractRay, recursionDepth + 1, refractDecay);
76                  // 如果没有阴影，或者有阴影但是阴影的距离大于光源到交点的距离，返回光照强度
77                  if ((!shadowHit) || (shadowHit && shadowHit->t > distance)) {
78                      plusRefLectRefrect = c * l.intensity
79                                         + reflectColor
80                                         + refractColor;
81                      return plusRefLectRefrect;
82                  }
83                  else {
84                      plusRefLectRefrect = Vec3{0}
85                                         + reflectColor
86                                         + refractColor;
87                      return plusRefLectRefrect;
```

```
                }
                break;
            }
            default:{
                // 如果没有阴影，或者有阴影但是阴影的距离大于光源到交点的距离，返回光照强度
                if ((!shadowHit) || (shadowHit && shadowHit->t > distance)) {
                    return c * l.intensity;// 没有遮挡，返回光照强度
                }
                else {
                    return Vec3{0};// 有遮挡，返回黑色
                }
            }
        }
    }
    else if(randomFloat < stopP){// 小于 stop 就直接终止递归
        return Vec3{0};// 根据概率终止递归
    }
}
// 如果没有击中物体，也是返回 rgb000--黑色
else {
    return {0, 0, 0};
}
```

（以上内容包含的 Dielectric 材质会在下文中介绍）

其实真正修改的地方是比想象中要少的，因为 Whitted-Style 算法和原有框架的 Ray-Cast 算法以及 SimplePathTracing 算法的许多代码文件都类似，所以一些定义就直接保留了下来，同时文件夹里面其他的相关文件也大差不差，基本上不需要大刀阔斧地修改，所以主要的算法实现部分都集中在了一个文件里面。此外，上述代码针对不同材质有不少相近的代码，其实可以进一步优化到一个单独的函数里面，但这里毕竟材质类别不多没什么必要，所以没有做进一步的封装，分别写出来显得更清晰一点。

(二) 俄罗斯轮盘赌

在上面的 Whitted-Style 递归光线追踪算法中，如果场景比较复杂的话，那么光线会不断进行递归，光线数量也会变得很庞大，让计算量增加，导致最后的渲染时间也增加，在这种情况下，如果我们还是按照设置好的 50 的递归深度的话，效果未必理想。

因此，我们想提高一些泛用性，尝试做一些优化，提高算法效率。在观看 Games101 和相关文章笔记之后，我们想到可以尝试引入俄罗斯轮盘赌，就是通过生成随机数来作为概率，让概率来决定递归是否终止，从而减少一些计算量，提高算法的效率。核心代码如下：

```
// 优化：加个俄罗斯轮盘赌，随机选择反射或折射
// 大致想法就是随机数低于我设定的值时就停止递归
float stopP = 0.0001f;// -> 直接终止递归的概率
// 随机数如果用 rand 总是生成 0.96 或者 0.97
```

```
// 查资料说是除以 MAX 的时候都趋于 0.96 了，不用 rand 了
// 找了个别的随机数库
static uniform_real_distribution<float> distribution(0.0f, 1.0f);
static default_random_engine generator;
float randomFloat = distribution(generator);
if (randomFloat < stopP) {
    return {0, 0, 0};// 根据概率终止递归
}
```

(三) Dielectric 材质

框架中已有的 Lambertian 材质模拟的是理想的漫反射，Phong 材质则包含漫反射和镜面反射，二者均没有折射，所以我们尝试新增了同时考虑反射和折射的 Dielectric 材质，从而让 Whitted-Style 算法在折射部分也有所体现。

仿照已有的材质新增 Dielectric.hpp 和 Dielectric.cpp，其中该材质定义如下;

```
// 加一个Dielectric材质，这个有折射，才能完整实现Whitted-style光线追踪算法
// 照搬另外2个材质的大致结构
namespace WhittedRayTracing
{
    class Dielectric : public Shader
    {
    private:
        float refractiveIndex;// 折射率

        Vec3 diffuseColor;// 漫反射颜色
        Vec3 reflectColor;// 反射颜色
        float reflectEx;// 反射指数
        Vec3 refractColor;// 折射颜色


    public:
        Dielectric(Material& material, vector<Texture>& textures);

        // virtual Refracted shade(const Ray& ray, const Vec3& hitPoint, const Vec3& normal) const;
        virtual RGB shade(const Vec3& in, const Vec3& out, const Vec3& normal) const;
    };
}
```

即将漫反射、反射、折射系数全都定义了出来，构造函数的部分直接模仿框架写即可，这里不再单独展示。

同时仿照路径追踪框架代码中存储光线散射信息的结构体，我也新增了一个存储反射和折射信息的结构体，如下:

```
namespace WhittedRayTracing
{
    // 用来放Dielectric材质的反射折射的变量
    struct Refracted
    {
        // 反射
        Ray refLectRay = {};
        Vec3 refLect_ratio = {};

        // 折射
        Ray refRectRay = {};
        Vec3 refRect_ratio = {};

    };// 分号

}
```

由于比之前多了折射，所以也仿照 Phong 中反射函数的实现，新增了折射函数的实现，以便 Whitted-Style 算法在过程中可以调用：

要编写折射函数，则需要通过数学和物理知识进行计算，我们初高中所学的斯涅尔定律告诉我们：

$$n_i \sin(\theta_i) = n_t \sin(\theta_t)$$

其中：

- n_i 是入射介质的折射率，n_t 是折射介质的折射率。
- θ_i 是入射角，θ_t 是折射角。

这只是一个简单的公式，在正式计算时，我们需要用到向量，参考网络资料以及 Games101，将其转化为向量形式如下：

$$\mathbf{T} = \eta \mathbf{I} + (\eta \cos\theta_i - \sqrt{1 - \eta^2(1 - \cos^2\theta_i)})\mathbf{N}$$

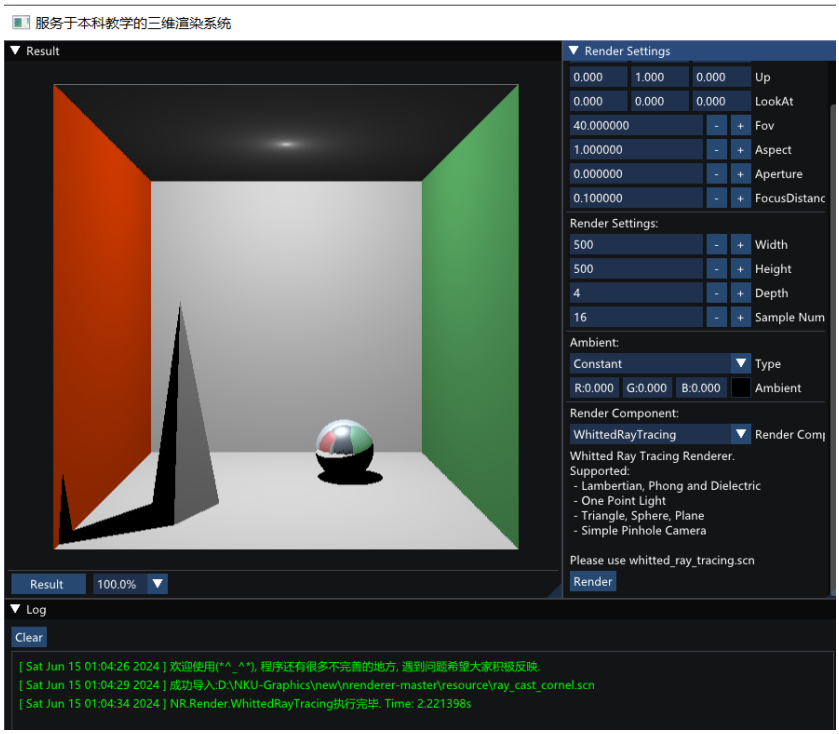
其中 $\mathbf{I}$ 是入射光线方向向量，$\mathbf{N}$ 是表面的法向量，$\eta = \frac{n_i}{n_t}$ 是两种介质的折射率之比。将其转化为代码如下：

```
Vec3 refract(const Vec3& I, const Vec3& N, float etaI, float etaT) {
    float cosi = glm::clamp(glm::dot(-I, N), -1.0f, 1.0f); // 入射角的余弦值
    float eta = etaI / etaT; // 折射率之比
    float k = 1.0f - eta * eta * (1.0f - cosi * cosi); // 计算 sin^2(_t)
    if (k < 0.0f) {
        return Vec3(0); // 全反射
    } else {
        return eta * I + (eta * cosi - sqrtf(k)) * N; // 计算折射方向的公式
    }
}
```

（最初尝试的过程中忽略了全反射，于是后来便直接丢给 ChatGPT 进行修改完善，生成公式对应的代码）

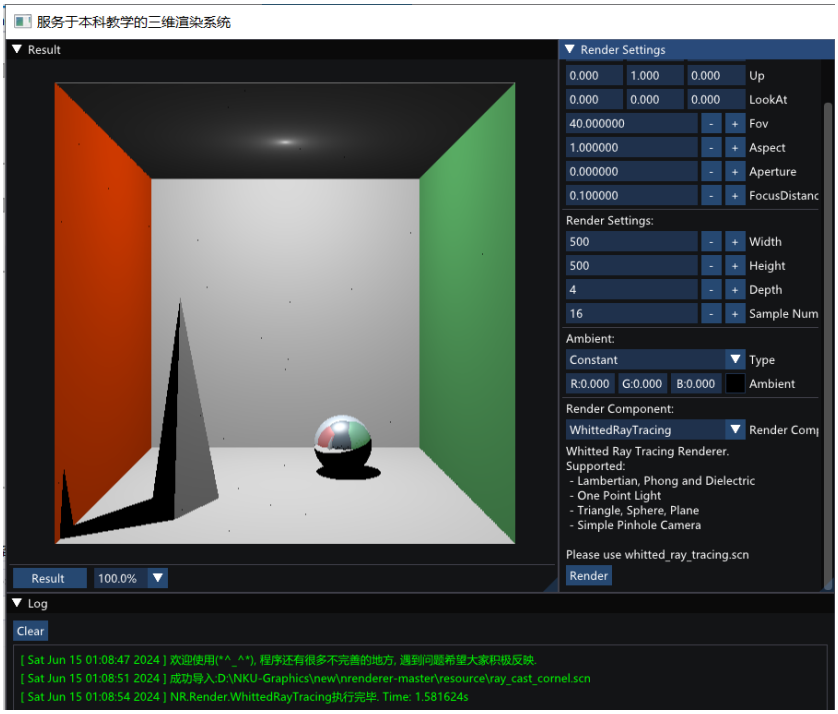
（四） 运行结果

1. Whitted-Style 递归光线追踪算法的结果



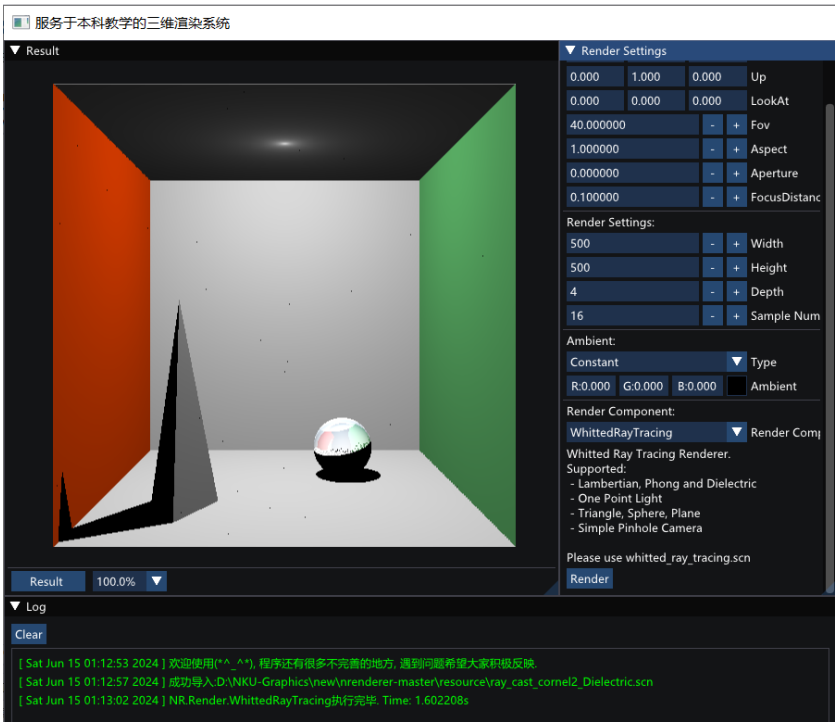
如图，Phong 材质的小球表面有了周围反射过来的颜色，效果比较理想，成功实现了 Whitted-Style 递归光线追踪算法。

2. 俄罗斯轮盘赌进行效率优化后的结果



（图中的有一些小黑点是因为俄罗斯轮盘赌的存在让递归提前终止了）可以看到，在生成结果基本一致的情况下，生成时间由 2.22s 减少至 1.58s，减小幅度约 28.8%，具备一定的优化效果。

3. 换材质的结果（未完全实现）



在新增了 Dielectric 材质并对场景文件做了一些修改之后的效果如图，可以看到结果并不理想，可能仍有部分文件我们没有修改到位。

三、 框架二

在流体模拟问题中，主要有两大算法：基于网格的欧拉模拟和基于粒子的拉格朗日模拟。拉格朗日视角关注粒子如何随着物理规律运动，而欧拉视角则关注空间中一组固定的网格点

(一) 欧拉模拟 (2D)

对于欧拉模拟，参考文章，可以得到相应的公式（考虑篇幅，只给出最终的公式，具体的化简和原理可以参考上述文章）：

$$\frac{D\mathbf{u}}{Dt} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\frac{1}{\rho}\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f} \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{2}$$

其中方程(1)为 Navier-Stokes 流体力学方程，方程(2)保证速度场散度为 0

对方程进一步处理如下：

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u}\cdot\nabla\mathbf{u} - \frac{1}{\rho}\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f} \tag{3}$$

对方程(3)分析，左侧即固定点上的加速度，右侧分别为对流项（$\mathbf{u}\cdot\nabla\mathbf{u}$）、压力项（$\frac{1}{\rho}\nabla p$）、粘度项（$\mu\nabla^2\mathbf{u}$）、外力项（$\mathbf{f}$），其中在本次实验中不考虑粘度项

按照上述文章和课程 PPT，具体的流体模拟步骤如下：

1. **Advection（平流）：**处理流体中的物质随速度场的移动。通过对速度场进行拉格朗日或欧拉方法的计算，使流体的属性（如速度、温度等）沿流场方向进行传输, 此处使用的对流算法为半拉格朗日方法
2. **External Forces（处理外力）：**考虑外力对流体的影响，此处计算的是温度差带来的浮力和烟雾所受的重力
3. **Projection（投影）：**为了保证流体的不可压缩性，需要确保速度场无散，等价于求解压力的泊松方程，来将速度场投影到一个无散度场

1. 平流

对于平流操作，此处选择使用半拉格朗日对流，即对于当前网格点 x 的数据，其速度值为 μ，则要计算当前时间的数据，可以后向追踪到上一个时间点的粒子位置 $x-\mu\Delta t$，即下图中的灰色粒子位置，然后对该位置的速度、密度等进行插值，得到当前网格点 x 的数据:

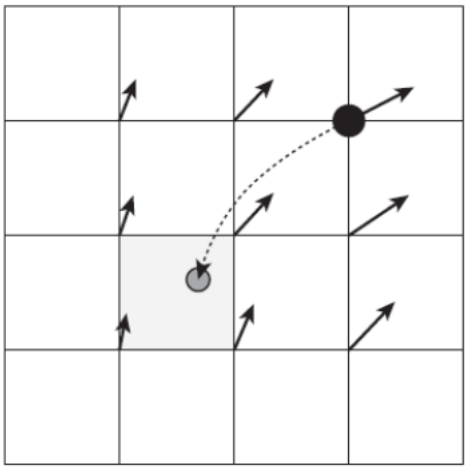


图 2: 半拉格朗日对流

具体的对流步骤为:

1. 密度和温度存储在网格中心点，因此获取网格中心点对应在上一个时间点的位置，得到插值数据，赋值给 newDensity 和 newTempeture
2. 速度 U 存储在网格的左边界，因此获取网格左边界在上一个时间点的位置，赋值给 newU
3. 速度 V 存储在网格的下边界，同理获得 newV

其中使用 newV、newU 的目的是避免直接更新数据会影响后续网格的插值

```
void advection(MACGrid2d& mGrid) {
    // 创建新的网格数据
    Glb::CubicGridData2d newDensity = mGrid.mD;
    Glb::CubicGridData2d newTempeture = mGrid.mT;
    Glb::GridData2dX newU = mGrid.mU;
    Glb::GridData2dY newV = mGrid.mV;

    // 遍历每个网格
    FOR_EACH_LINE {
        // 密度和温度的更新
        glm::vec2 pt = mGrid.getCenter(i, j); // 获取当前单元中心点
        // 通过半拉格朗日方法获取前一个位置
        glm::vec2 lastPos = mGrid.semiLagrangian(pt, Lagrangian2dPara::dt);
        newDensity(i, j) = mGrid.mD.interpolate(lastPos); // 插值计算新的密度值
        newTempeture(i, j) = mGrid.mT.interpolate(lastPos); // 插值计算新的温度值

        // 速度 U 的更新，存储在左边界
        if (i < mGrid.dim[0] && j < mGrid.dim[1]) {
            glm::vec2 pt = mGrid.getLeft(i, j); // 获取左边界点
            // 获取前一个位置
```

```
            glm::vec2 lastPos = mGrid.semiLagrangian(pt, Lagrangian2dPara::dt);
            newU(i, j) = mGrid.getVelocityX(lastPos); // 插值计算新的 x 方向速度
        }

        // 速度 V 的更新，存储在下边界
        if (i < mGrid.dim[0] && j < mGrid.dim[1]) {
            glm::vec2 pt = mGrid.getBottom(i, j); // 获取下边界点
            // 获取前一个位置
            glm::vec2 lastPos = mGrid.semiLagrangian(pt, Lagrangian2dPara::dt);
            newV(i, j) = mGrid.getVelocityY(lastPos); // 插值计算新的 y 方向速度
        }
    }

    // 更新网格数据
    mGrid.mU = newU;
    mGrid.mV = newV;
    mGrid.mD = newDensity;
    mGrid.mT = newTempeture;
}
```

其中原始框架提供了两个宏：FOR_EACH_LINE 和 FOR_EACH_CELL，一个用于遍历网格边界，一个用于遍历网格中心，其中由于框架代码确保了数据的正确访问，因此对于网格中心的遍历也可以使用网格边界的遍历宏

2. 处理外力

对于本次的烟雾模拟来说，烟雾粒子所受的外力主要有重力、烟雾和环境之间温度差异引起的力，两个组合成 BoussinesqForce，公式为：

$$\mathbf{f}_{\text{buoy}} = -\alpha\rho + \beta(T - T_{\text{amb}}) \tag{4}$$

其中相关的代码如下：

```
// 根据温度差异计算 Boussinesq Force
double MACGrid2d::getBoussinesqForce(const glm::vec2 &pos)
{
    // f = [0, -alpha*smokeDensity + beta*(T - T_amb)]
    double temperature = getTemperature(pos);
    double smokeDensity = getDensity(pos);

    double yforce = -Eulerian2dPara::boussinesqAlpha * smokeDensity +
                    Eulerian2dPara::boussinesqBeta *
                    (temperature - Eulerian2dPara::ambientTemp);

    return yforce;
}
```

```
14
15   //施加外力的影响
16   void external_forces(MACGrid2d& mGrid) {
17       FOR_EACH_LINE{
18           glm::vec2 center=mGrid.getCenter(i, j);
19           mGrid.mV(i, j) += mGrid.getBoussinesqForce(center) * Lagrangian2dPara::dt;
20       }
21   }
```

3. 散度

在对速度场进行投影之前，需要先了解散度的概念。散度衡量的是某个点周围向量场的流入和流出量的差异，对于大多数情况下的流体（如本次实验中的烟雾流体），流体体积变化很小，由 [Yang 2019] 的文章流体模拟 Fluid Simulation：流体模拟基础可知，流体的不可压缩性等价于公式(2)，即速度场无散

散度计算公式：

$$\nabla \cdot \mathbf{u} = \nabla \cdot (u, v) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{5}$$

$$\Delta u = u_{i+1,j} - u_{i,j} \tag{6}$$

$$\Delta v = v_{i,j+1} - v_{i,j} \tag{7}$$

即

$$\nabla \cdot \mathbf{u} \approx \frac{\Delta u}{\Delta x} + \frac{\Delta v}{\Delta y} \approx \frac{u(i+1,j) - u(i,j)}{\text{cellSize}} + \frac{v(i,j+1) - v(i,j)}{\text{cellSize}} \tag{8}$$

对应的代码如下：

```
1   double MACGrid2d::checkDivergence(int i, int j)
2   {
3       double x1 = mU(i + 1, j);
4       double x0 = mU(i, j);
5       double y1 = mV(i, j + 1);
6       double y0 = mV(i, j);
7       double xdiv = x1 - x0;
8       double ydiv = y1 - y0;
9       double div = (xdiv + ydiv) / cellSize;
10      return div;
11  }
```

4. 投影

由文章可知，速度场的散度为 0 可以等价于求解下列的压力方程：

$$\frac{\Delta t}{\rho} \nabla^2 p = \nabla \cdot \mathbf{u}^n \tag{9}$$

其中，$\nabla^2 p$ 是压力的拉普拉斯算子，$\nabla \cdot \mathbf{u}^n$ 是速度场的散度。

对于网格单元 (i, j)，离散化泊松方程可以表示为：

$$\nabla^2 p \approx \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2} \tag{10}$$

将其代入泊松方程，得到：

$$\frac{\Delta t}{\rho} \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\Delta x)^2} = \nabla \cdot \mathbf{u}^n \tag{11}$$

简化得到：

$$p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j} = \frac{\rho(\Delta x)^2}{\Delta t} \nabla \cdot \mathbf{u}^n \tag{12}$$

为了求解上述方程，使用迭代方法更新压力值。假设当前迭代步的压力值为 p，在下一迭代步的新压力值为 p^{new}，更新公式为：

$$p_{i,j}^{\text{new}} = \frac{\frac{-\rho(\Delta x)^2}{\Delta t} \nabla \cdot \mathbf{u}^n + p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1}}{4} \tag{13}$$

如果考虑网格是固体的话，公式计算的应该是非固体网格的压力：

$$p_{i,j}^{\text{new}} = \frac{\frac{-\rho(\Delta x)^2}{\Delta t} \nabla \cdot \mathbf{u}^n + \sum_{\text{neighbors}} p_{\text{valid}}}{\text{number of valid neighbors}} \tag{14}$$

上述公式存在着一定的数值散耗，因此通过迭代多次尽可能使得压力的求解趋于正确值（此处迭代 50 次或者所有压力迭代的残差最大不能大于 1e-6）

一些变量的定义：

```
int iterations = 50; // 最大迭代次数
double dt = Lagrangian2dPara::dt; // 时间步长
double dx = mGrid.cellSize; // 空间步长
double p = 1.3;//空气密度
double alpha = -p * dx * dx / dt; // 迭代系数，即先前公式的散度前面的系数
double tolerance = 1e-6; // 收敛阈值
```

压力的计算：

```
// 迭代 50 次
for (int k = 0; k < iterations; k++) {
    double maxResidual = 0.0; // 最大残差
    std::vector<std::vector<double>> newPressure = pressure; // 计算新的压力

    FOR_EACH_CELL {
        if (mGrid.isSolidCell(i, j)) continue; // 跳过固体单元
        double div = mGrid.checkDivergence(i, j); // 获取当前位置的散度
        double p_neighbors = 0.0; // 旁边网格的压力
        int num_neighbors = 0; // 非固定网格的数量

        // 计算相邻网格单元的压力值 p(i,j+1) 和压力系数
        if (j + 1 < Eulerian2dPara::theDim2d[MACGrid2d::Y]
```

```
            && !mGrid.isSolidCell(i, j + 1))
        {
            p_neighbors += pressure[i][j + 1];
            num_neighbors++;
        }
        // 后续同理
        if (j - 1 >= 0 && !mGrid.isSolidCell(i, j - 1)) {}
        if (i + 1 < Eulerian2dPara::theDim2d[MACGrid2d::X]
            && !mGrid.isSolidCell(i + 1, j)) {}
        if (i - 1 >= 0 && !mGrid.isSolidCell(i - 1, j)) {}

        if (num_neighbors > 0) {
            newPressure[i][j] = (div * alpha + p_neighbors) / num_neighbors;
        }

        // 计算网格单元的残差，并更新最大残差
        double residual = fabs(newPressure[i][j] - pressure[i][j]);
        if (residual > maxResidual) {
            maxResidual = residual;
        }
    }

    pressure = newPressure; // 更新压力值

    // 检查收敛条件，如果最大残差小于容差，则停止迭代
    if (maxResidual < tolerance) { break; }
}
```

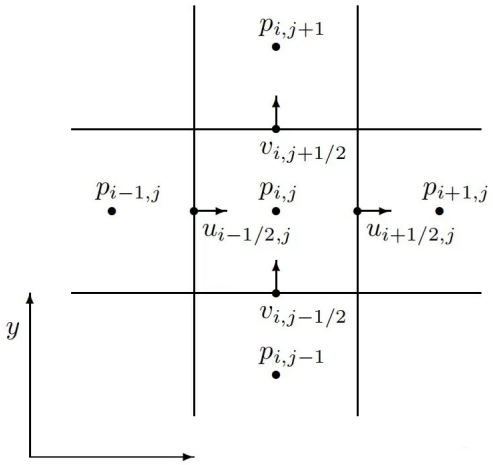


图 3: 压力值存储

最后如上图所示，对于每个非固体单元，根据压力梯度更新速度的物理公式如下：

1. 更新水平速度分量 u:

$$u(i,j) = u(i,j) - \frac{\Delta t}{\rho \Delta x}(p(i,j) - p(i-1,j)) \tag{15}$$

2. 更新垂直速度分量 v:

$$v(i,j) = v(i,j) - \frac{\Delta t}{\rho \Delta y}(p(i,j) - p(i,j-1)) \tag{16}$$

```
FOR_EACH_CELL {
    if (!mGrid.isSolidCell(i,j)) {
        // 更新速度
        double p_x = 0.0;
        double p_y = 0.0;

        if (i - 1 >= 0 && !mGrid.isSolidCell(i - 1, j)) {
            p_x = pressure[i][j] - pressure[i - 1][j];
        }

        if (j - 1 >= 0 && !mGrid.isSolidCell(i, j - 1)) {
            p_y = pressure[i][j] - pressure[i][j - 1];
        }

        mGrid.mU(i, j) -= dt / (p * dx) * p_x;
        mGrid.mV(i, j) -= dt / (p * dx) * p_y;
    }
}
```

5. 运行结果

通过上述步骤，运行结果如下:

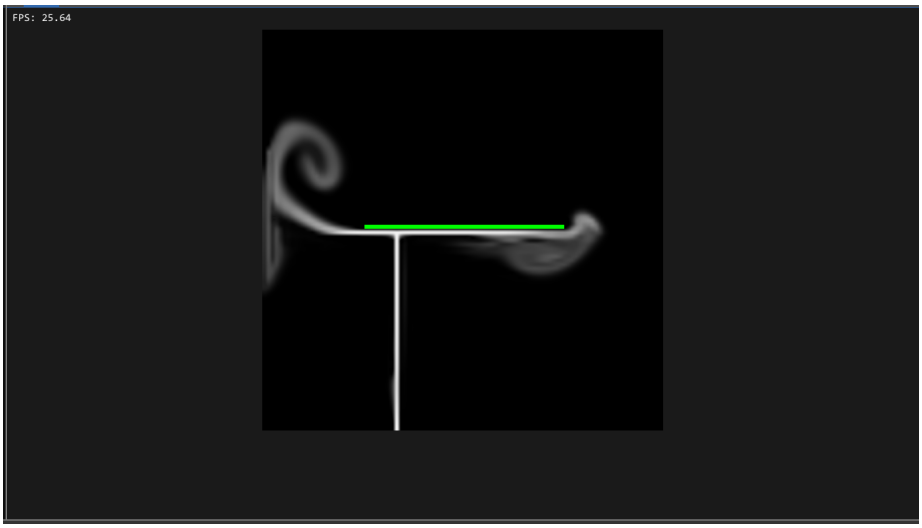


图 4: 欧拉流体模拟 (2D)

(二) 欧拉模拟 (3D)

三维情况下的欧拉模拟和二维情况下的类似，只需要增加一个维度就行，以平流为例：

```
// 创建新密度和温度网格数据
Glb::CubicGridData3d newDensity = mGrid.mD;
Glb::CubicGridData3d newTemperature = mGrid.mT;

// 创建新的速度网格数据
Glb::GridData3dX newU = mGrid.mU;
Glb::GridData3dY newV = mGrid.mV;
Glb::GridData3dZ newW = mGrid.mW;
```

进行半拉格朗日对流，其中此处 x 方向速度是存储在后边界，y 方向速度是存储在左边界，z 方向速度存储在下边界：

```
for (int i = 0; i < mGrid.dim[0]; ++i) {
    for (int j = 0; j < mGrid.dim[1]; ++j) {
        for (int k = 0; k < mGrid.dim[2]; ++k) {
            glm::vec3 pt = mGrid.getCenter(i, j, k); // 获取当前单元中心点
            // 通过半拉格朗日方法获取前一个位置
            glm::vec3 lastPos = mGrid.semiLagrangian(pt,
                                                     Lagrangian3dPara::dt);
            // 插值计算新的密度值
            newDensity(i, j, k) = mGrid.mD.interpolate(lastPos);
            // 插值计算新的温度值
            newTemperature(i, j, k) = mGrid.mT.interpolate(lastPos);

            if (i < mGrid.dim[0]) {
                glm::vec3 pt = mGrid.getBack(i, j, k); // 获取后边界点
                // 获取前一个位置
                glm::vec3 lastPos = mGrid.semiLagrangian(pt,
                // 插值计算新的 x 方向速度
                newU(i, j, k) = mGrid.getVelocityX(lastPos);
            }

            if (j < mGrid.dim[1]) {
                glm::vec3 pt = mGrid.getLeft(i, j, k); // 获取左边界点
                // 获取前一个位置
                glm::vec3 lastPos = mGrid.semiLagrangian(pt,
                                                         Lagrangian3dPara::dt);
                // 插值计算新的 y 方向速度
                newV(i, j, k) = mGrid.getVelocityY(lastPos);
            }
```

```
29
30              if (k < mGrid.dim[2]) {
31                  glm::vec3 pt = mGrid.getBottom(i, j, k); // 获取下边界点
32                  // 获取前一个位置
33                  glm::vec3 lastPos = mGrid.semiLagrangian(pt,
34                                                  Lagrangian3dPara::dt);
35                  // 插值计算新的 z 方向速度
36                  newW(i, j, k) = mGrid.getVelocityZ(lastPos);
37              }
38          }
39      }
40  }
```

最后更新速度场和网格数据:

```
1   // 更新速度场
2   mGrid.mU = newU;
3   mGrid.mV = newV;
4   mGrid.mW = newW;
5
6   // 更新网格数据
7   mGrid.mD = newDensity;
8   mGrid.mT = newTemperature;
```

其余步骤同理，只需要额外处理一个维度，最终运行结果如下:

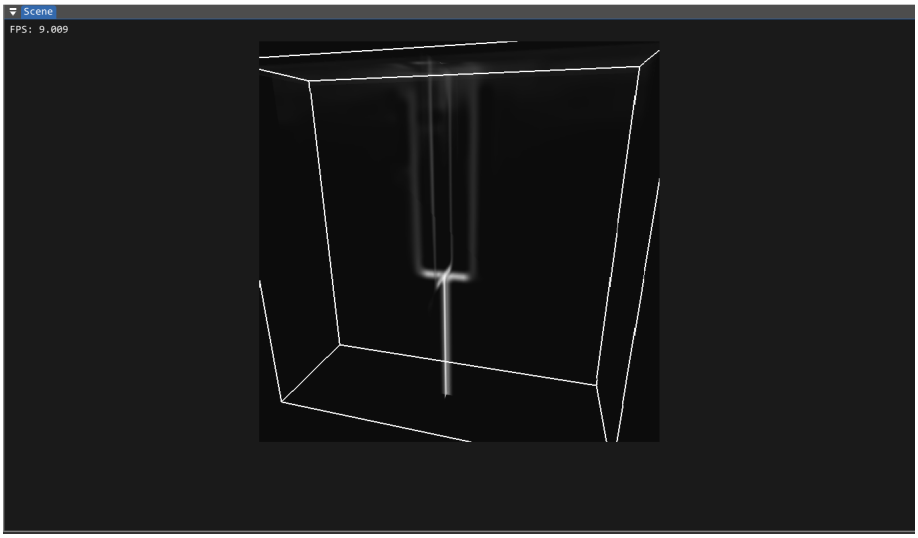


图 5: 欧拉流体模拟 (3D)

(三) 拉格朗日模拟 (2D)

基于拉格朗日的流体模拟中，此处选择使用 SPH 的拉格朗日流体模拟算法，具体步骤有:

1. **初始化：**初始化粒子的质量等
2. **更新粒子密度：**根据粒子的位置和邻域内的其他粒子，重新计算每个粒子的密度。使用核函数计算密度
3. **对粒子施加重力：**对每个粒子施加重力
4. **对粒子施加压力：**计算每个粒子的压力，并施加压力梯度力
5. **对粒子施加粘性力：**计算并施加粘性力，模拟流体的内部摩擦
6. **计算加速度，更新粒子的速度、位置，处理碰撞：**计算每个粒子的总力，并计算加速度。使用更新后的加速度和速度，更新每个粒子的位置信息。检查并处理粒子与边界或障碍物的碰撞，根据碰撞条件调整粒子的速度和位置

1. 初始化

此处进行一些初始化操作，方便后面的计算和处理，如后续会经常用到支持半径的平方，可以避免每一个粒子更新的时候都要重新算一次，从而减少时间开销：

```
Solver::Solver(ParticleSystem2d& ps) : mPs(ps)
{
    // 计算粒子的质量：粒子的体积乘以流体的密度
    mass = mPs.particleVolume * Lagrangian2dPara::density;

    // 支持半径 h 可以是支持半径 sR 加上粒子半径 pR，或者仅为支持半径 sR
    // 两者模拟结果差别不大，h 较大时粒子之间间隔比较大
    // h = mPs.supportRadius;
    h = mPs.supportRadius + mPs.particleRadius;  // 当前选择支持半径加粒子半径

    // 支持半径的平方
    h2 = mPs.supportRadius2;

    // 支持半径的三次方，用于加速计算
    h3 = h * h2;

    // 粒子质量的平方，用于后续计算
    mass_2 = std::pow(mass, 2);

    // 网格块的最大数量：网格在 x 方向的块数乘以 y 方向的块数
    max_Num = (double)mPs.blockNum.x * (double)mPs.blockNum.y;
}
```

2. 邻域粒子搜索

在拉格朗日流体模拟中，后续粒子密度、所受压力等的计算都要使用邻域粒子信息，因此先实现获取邻域粒子信息的函数：

对于某个粒子，获取其所在位置的区块 ID：

```
// 获取粒子所在位置的区块 ID
uint32_t blockId = mPs.getBlockIdByPosition(p.position);

std::vector<FluidSimulation::Lagrangian2d::ParticleInfo2d> blockParticles;

if (blockId < 0) { // 找不到对应的 ID
    return blockParticles;
}
```

当前区块 ID 的相邻区块偏移存储在框架代码中的 blockIdOffs 中，因此可以通过 ID 减去相应的偏移访问相邻的区块，并对相邻区块的每一个粒子，判断和当前粒子的距离是否小于支持半径 h:

```
for (auto& index : mPs.blockIdOffs) {
    uint32_t block_temp = blockId - index;

    // 如果区块 ID 无效，跳过该区块
    if (block_temp < 0 || block_temp > max_Num) {
        continue;
    }

    // 获取当前区块的粒子范围
    glm::vec2 range = mPs.blockExtens[block_temp];
    size_t startIndex = range.x;
    size_t endIndex = range.y;

    // 遍历区块中的所有粒子
    for (int i = startIndex; i < endIndex; i++) {
        try {
            // 计算当前粒子与目标粒子的距离
            double distance = glm::length(mPs.particles[i].position
                                          - p.position);

            // 如果距离小于等于支持半径，则将该粒子添加到附近粒子向量中
            if (distance <= h) {
                blockParticles.push_back(mPs.particles[i]);
            }
        }
        catch (const std::exception& e) {
           //输出一些错误信息
        }
    }
}
```

```
31
32  return blockParticles;
```

最终通过调用上述代码，可以获取和当前粒子距离小于等于 h 的粒子

3. 更新粒子密度

对于粒子密度的更新，可以参考流体模拟 Fluid Simulation：基于 SPH 的拉格朗日流体模拟:

$$f(\mathbf{x}) \approx \sum_{i=1}^{N} m_i w\left(\frac{||\mathbf{x}-\mathbf{x}_i||}{h_i}\right) \tag{17}$$

其中使用 poly6 核函数计算 $w\left(\frac{||\mathbf{x}-\mathbf{x}_i||}{h_i}\right)$:

$$W(\mathbf{r},h) = \begin{cases} \frac{4}{\pi h^8}(h^2-\mathbf{r}^2)^3 & \text{if } 0 \le \mathbf{r} \le h, \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

由于此次实验设定每个粒子的质量都相等，因此当前粒子的密度等于邻域粒子的核函数之和再乘上质量:

```
1   //更新密度
2   void update_D(FluidSimulation::Lagrangian2d::ParticleSystem2d& mPs) {
3       //更新每一个粒子的密度
4       for (auto& p : mPs.particles) {
5           //获取领域粒子
6           blockParticles = getNearBy(p, mPs);
7
8           if (blockParticles.size() != 0) {
9               //计算邻域粒子核函数之和
10              double sum = sumOfNearby(p, blockParticles);
11              //当前粒子的密度等于核函数之和 * 质量
12              p.density = mass * sum;
13          }
14      }
15  }
```

poly6 核函数的代码:

```
1   inline double SphKernel(double distance) {
2       if (distance * distance >= h2)
3           return 0.0;
4       else {
5           double result = 4.0 / (M_PI * std::pow(h2, 4)) *
6                           std::pow(h2 - distance * distance, 3);
7           return result;
```

```
    }
}
```

核函数求和的代码:

```
double sum = 0.0;
for (auto& p : blockParticles) {
    double distance = glm::length(p.position - particle.position);
    sum += SphKernel(distance);
}
return sum;
```

4. 对粒子施加重力

在拉格朗日视角中，流体以粒子形式运动，会受到外部力，如重力、风力等，其中本次实验中只考虑重力因素，只需要在竖直方向 (2 维情况下为 y 方向) 加上重力即可:

```
//重力 =g*mass
p.force = glm::vec2(0.0f, -Lagrangian2dPara::gravityY * mass);
```

5. 对粒子施加压力

对于不可压缩流体，按照先前文章，选用泰特的状态方程:

$$P = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{19}$$

公式 (19) 中，ρ_0 是流体静止时的密度 (此处设为 1000)，ρ 是流体粒子当前的密度，γ 是状态方程的指数，取值为 7，而 B 则是一个缩放系数因子，按照框架代码为变量 stiffness:

```
double computePressure(const ParticleInfo2d& particle) {
        double p = stiffness *
                (std::pow(1.0 * particle.density / (1.0 * 1000), 7) - 1.0f);
        if (p < 0)return 0;
        return p;
    }
```

计算粒子所受压力的公式:

$$\mathbf{f}_p = -m^2 \sum_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \nabla W(\mathbf{x} - \mathbf{x}_j) \tag{20}$$

```
//周围粒子
blockParticles = getNearBy(p, mPs);

//压力
```

```
if (blockParticles.size() != 0) {
    for (auto& nearby : blockParticles) {
        if (&p != &nearby) {
            glm::vec2 r = p.position - nearby.position;
            double temp = mass_2 * (p.pressDivDens2 + nearby.pressDivDens2);
            p.force -= glm::vec2(temp) * poly6KernelGradient2D(r, h);
        }
    }
}
```

其中上述代码中的 poly6KernelGradient2D(r, h) 即 $W(\mathbf{x}-\mathbf{x}_j)$，其中具体的公式没找到相关资料，参考三维情况下压力更新可知，$W(\mathbf{x}-\mathbf{x}_j)$ 等价于对 r 求一阶偏导：

$$\nabla W(\mathbf{r},h)=\begin{cases}-24\frac{\mathbf{r}}{\pi h^8}(h^2-\mathbf{r}^2)^2 & \text{if } 0\le\|\mathbf{r}\|\le h,\\ 0 & \text{if } \|\mathbf{r}\|>h\end{cases} \tag{21}$$

```
// 二维 Poly6 核函数梯度实现
glm::vec2 poly6KernelGradient2D(const glm::vec2& r, float h) {
    float h2 = h * h; // 平滑长度的平方
    float r2 = glm::dot(r, r); // 计算距离的平方
    if (r2 > h2) return glm::vec2(0.0f, 0.0f);
    // 如果距离大于平滑长度，则返回 0 向量
    float hr2 = h2 - r2; // 计算 (h^2 - r^2)
    float coef = -24.0f / (M_PI * std::pow(h, 8)); // 计算常数系数
    return coef * hr2 * hr2 * r; // 返回梯度值
}
```

6. 对粒子施加粘性力

粘性力的计算公式参考了论文里的公式 (14)

$$\mathbf{f}_v(\mathbf{x})=m\mu\sum_j\left(\frac{\mathbf{v}_j-\mathbf{v}_i}{\rho_j}\right)\nabla^2W(\mathbf{x}-\mathbf{x}_j) \tag{22}$$

$\nabla^2W(\mathbf{x}-\mathbf{x}_j)$ 的计算如下（仿照三维的拉普拉斯算子的计算）：

```
double Laplacian(double distance) {
    if (h < distance) {
        return 0;
    }
    return 30.0 / (M_PI * h2 * h2 * h) * (h - distance);
    //下面这个是 Poly6 求导得来的
    //return 24 / (M_PI * std::pow(h2, 4)) *
    //(h2 - 5 * distance * distance) * (h2 - distance * distance);
}
```

最后粘性力的更新如下:

```
//粘性力
if (blockParticles.size() != 0) {
    for (auto& nearby : blockParticles) {
        if (&p != &nearby) {
            double distance = glm::length(p.position - nearby.position);
            double temp = viscosity * (double)mass *
                        Laplacian(distance) / nearby.density;
            p.force += glm::vec2(temp) * (nearby.velocity - p.velocity);
        }
    }
}
```

7. 更新粒子速度、位置等

处理粒子受到的所有力后，计算粒子的加速度，并以此加速度对粒子的速度进行更新:

```
//加速度
p.acceleration = p.force / mass;

//速度
p.velocity += p.acceleration * Lagrangian2dPara::dt;
```

对于粒子的位置，则需要考虑是否和墙壁发生碰撞:

```
//计算新的位置
glm::vec2 new_position = p.position + p.velocity * Lagrangian2dPara::dt;

//确保不发生穿透，若碰撞到墙壁，需要对粒子的新位置进行更新
//同理，也可以对粒子的速度进行相应的处理，此处简化为碰撞后速度为 0
if (new_position.y <= mPs.lowerBound.y) {
     new_position.y = mPs.lowerBound.y + Lagrangian2dPara::eps;
     p.velocity.y = 0;
}
//其余同理
```

最后更新粒子的位置和区块 ID:

```
//更新粒子位置
p.position = new_position;

//更新粒子所属的块 ID
p.blockId = mPs.getBlockIdByPosition(p.position);
```

8. 运行结果

通过上述处理后，运行结果如下：

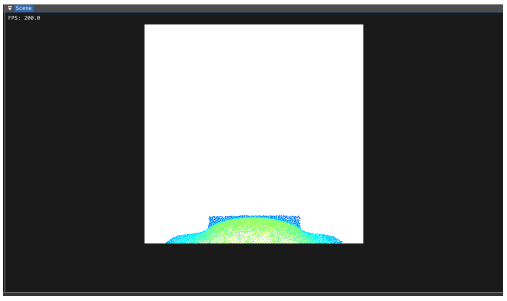


图 6: 拉格朗日流体模拟 (1)

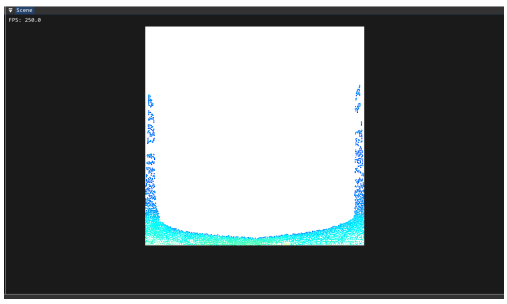


图 7: 拉格朗日流体模拟 (2)

(四) 拉格朗日模拟 (3D)

三维情况下的拉格朗日流体模拟处理步骤类似，主要的区别在三维的拉格朗日使用的函数和二维的存在差异，具体体现为：

1. 粒子更新密度时使用的：

$$W_{poly6}(r,h) = \frac{315}{64\pi h^3} \begin{cases} \left(1 - \left(\frac{r^2}{h^2}\right)\right)^3 & \text{if } 0 \le r \le h \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

2. 粒子更新压强时使用的：

$$\nabla W_{spiky}(r,h) = -\frac{45}{\pi h^4}\left(1 - \frac{r}{h}\right)^2 \tag{24}$$

3. 粒子更新粘性力时使用的：

$$\nabla^2 W(r,h) = \frac{45}{\pi h^6}(h - r) \tag{25}$$

篇幅所限，此处只给出运行结果：

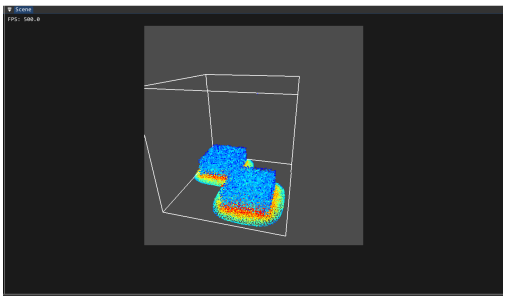


图 8: 流体碰到墙壁开始散开

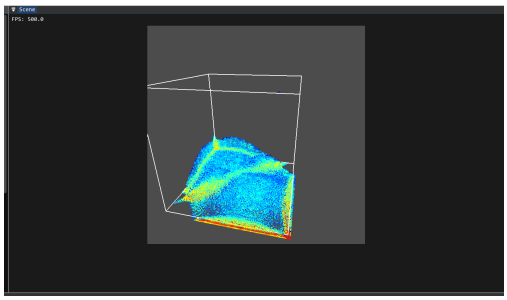


图 9: 四周的粒子溅起

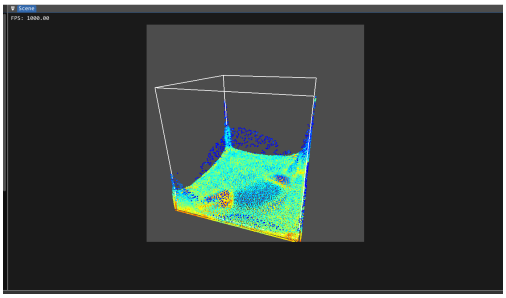


图 10: 流体再次碰到墙壁

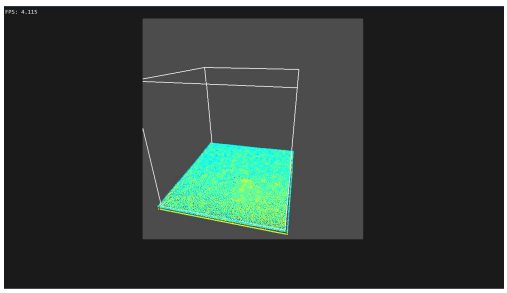


图 11: 最终粒子停止运动

(五) 并行处理

完成上述内容后，本次实验还对欧拉模拟和拉格朗日尝试进行并行处理，示例如下:

```
#define NUM_THREADS 4

// 并行处理速度场的对流更新
#pragma omp parallel for num_threads(NUM_THREADS)
for (int j = 0; j < Eulerian2dPara::theDim2d[MACGrid2d::Y] + 1; j++)
    for (int i = 0; i < Eulerian2dPara::theDim2d[MACGrid2d::X] + 1; i++){
        //速度的更新，和先前一样
    }
```

其中通过测试不同线程数情况下二维烟雾碰撞到容器中心的固体隔板所用的时间, 结果表明时间差距不大，一方面原因可能在于每一步模拟的时候压力的更新迭代（最多迭代 50 次）占大部分时间，而下一次压力的计算又需要上一次压力的数据，因此大部分情况下都是在串行执行，另一方面通过相同的方法对拉格朗日流体进行并行处理得到的结果差距也不大, 因此更可能的原因是 OpenMP 的并行不成功，后续可以尝试使用 Thread 库手动编写多线程

(六) 结果对比

对于拉格朗日流体模拟, 还可以考虑使用的不同的核函数, 如先前的压力计算使用的是 Spiky 核函数，此处可以使用 Poly6 核函数更新压力:

$$\nabla W_{\text{poly6}}(r,h) = \begin{cases} -945\frac{r}{32\pi h^5}\left(1-\frac{r^2}{h^2}\right)^2, & 0 \le r \le h \\ 0, & r > h \end{cases} \tag{26}$$

除此之外，对于粒子碰撞墙壁之后，原始处理中直接将法线方向上的速度设为 0，此处可以选择修改为衰减到原来的-0.7 倍

部分结果示例如下:

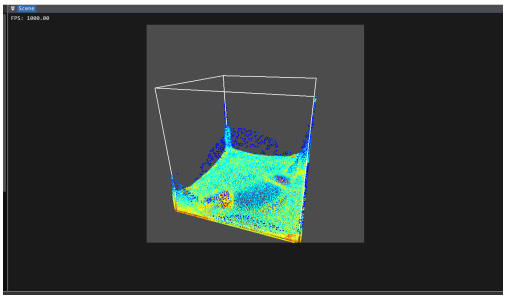


图 12: Spiky 核函数

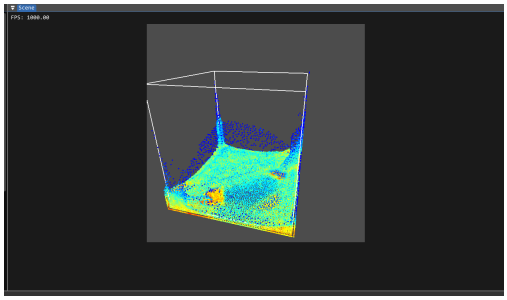


图 13: Poly6 核函数

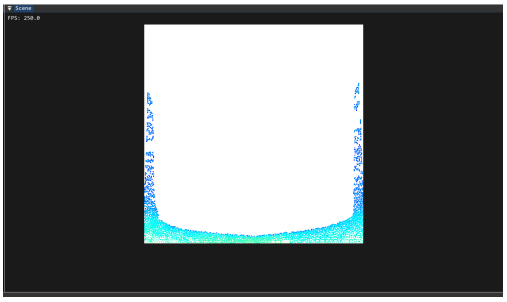


图 14: 速度衰减为 0

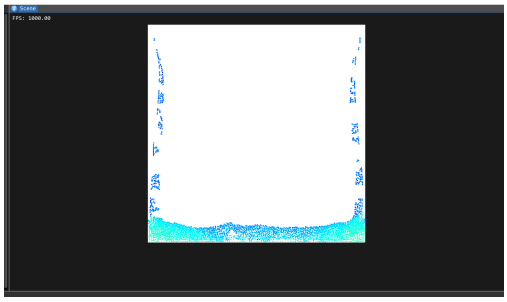


图 15: 速度衰减为-0.7 倍

其中运行结果差距都不大。

四、 总结与感想

- 经历了本次图形学大作业，我们首先是对于如何处理庞大的代码框架有了更多的经验，首先控制住自己的畏难情绪，大胆开始看，从 main 函数开始，看不懂的地方借助 ChatGPT 和 Github Copilot 都能够有效解决，其次是直接从需要实现的文件夹入手，其他文件可以暂且搁置，一旦把要实现的算法文件夹中的文件结构梳理一遍，就会发现其实很多文件都是重复的，大同小异的，真正的核心文件很少甚至只有一个最核心的算法文件。而在需要实现的文件夹中，点击某些变量或者函数都可以直接跳转到定义，从而可以对框架代码中其他部分更加熟悉，在后续编写代码的过程中也能知道框架中定义的部分函数如何使用。

- 在渲染框架的实现中，我们对于光线追踪算法的整体发展流程也有了更多的了解，明白了一个逼真的图背后的生成原理，也在自己实现 Whitted-Style 算法的过程中对于反射、折射等有了更深的理解，虽然 Dielectric 材质的实现由于时间有限还存在一些问题，但我们也从中对于不同材质的实现区别有了更多认知，获得了更多的实践经验。

- 对于流体框架的实现，更是历经波折。在写流体之前，笔者先完整学习了 games101 的所有视频，并完成了作业 0——6 的所有基础要求和提高要求（作业 7 和 8 水了，完整的 games101 作业在github上），接着为了实现流体，还观看了games201的前 7 讲和games103的最后两章。流体算法的实现相比于渲染，一个重大的特点就是流体渲染的公式不好找，甚至几乎找不到完整的 2 维情况下的欧拉和拉格朗日的物理公式，有时候经常错误的把三维的公式应用到二维上，当时将三维的压力公式应用到二维上时，一运行粒子就爆炸飞出边界，更是调了几天才解决问题。不过完整的实现下来，这次的实验还是大大锻炼了我的编码能力和一行代码调一天的 debug 能力 QAQ。

参考文献

[1] Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, pages 4–es, 2005.

[2] Jiří Vorba and Jaroslav Křivánek. Adjoint-driven Russian roulette and splitting in light transport simulation. *ACM Transactions on Graphics*, 35(4):1–11, 2016.

[3] https://blog.csdn.net/qq_37366618/article/details/123504877

[4] https://blog.csdn.net/qq_38065509/article/details/106299336

[5] https://blog.csdn.net/Motarookie/article/details/122425896

[6] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '03)*, pages 154–159, Goslar, DEU, 2003. Eurographics Association.

[7] https://yangwc.com/2019/08/29/SPH/

[8] https://www.cnblogs.com/Ligo-Z/p/16226448.html

[9] https://yangwc.com/2019/09/12/Smoke/

[10] https://yangwc.com/2019/08/03/MakingFluidImcompressible/

[11] https://yangwc.com/2019/05/01/fluidSimulation/

[12] https://zhuanlan.zhihu.com/p/261927963