计算机图形学
第六次实验报告

MassSpring

PB20000264
韩昊羽

一. 实验要求

- 学会物理仿真的基本方法
- 实现欧拉半隐式方法
- 实现欧拉隐式方法
- 学会使用牛顿法解方程
- 实现欧拉隐式方法加速

二. 操作环境

IDE：Microsoft Visual Studio 2019 community

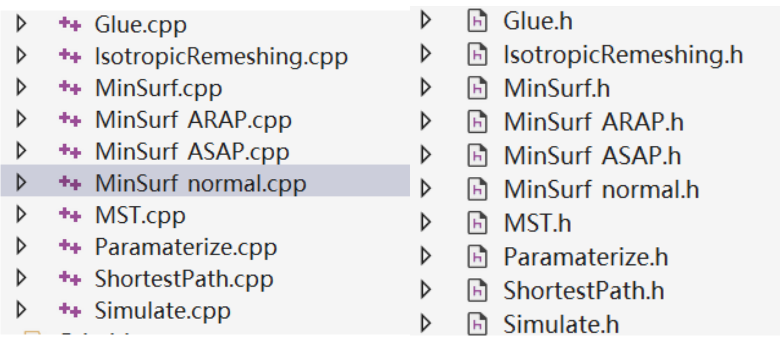
QT： 5.12.12

Cmake：3.23.1

UEngine

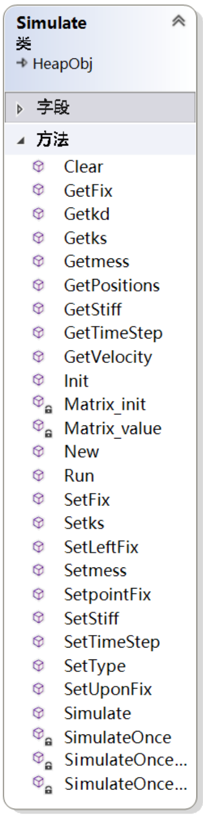
三. 架构设计

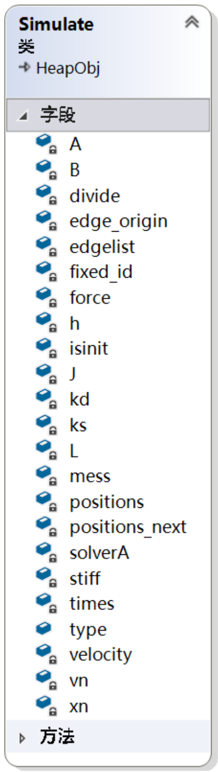
3.1 文件结构



相比于上一次作业没有改动，全部功能在 simulate 类中实现。

3.2 类图





对于字段，A,B,J,L 是矩阵，divide 是对步长进行缩小的倍数，edge_origin 存了原长，kd,ks,mess 是参数，times 是加速方法迭代的次数，xn,vn 标记上一帧的值。

对于方法，get 和 set 方法大多都是设置参数，更改参数的接口，Matrix_init 是加速算法中矩阵的初始化，matrix_value 是九宫格矩阵赋值。最后三个 Simulate 开头的方法是用三种不同方法进行模拟。

四. 功能实现

对于物理仿真系统，基本的思路是根据上一帧的位移和速度，通过物理公式推导出这一帧的位移和速度，并进行更新，以此迭代。

欧拉半隐式方法：

欧拉半隐式方法核心是通过上一帧的位移和速度推导出这一帧的力，通过牛顿第二定律得出加速度，再更新速度，得出位移，并以此迭代。基本公式为：

$$v_{n+1} = v_n + hM^{-1}f(x_n)$$

$$x_{n+1} = x_n + hv_{n+1}$$

流程如下：

Algorithm 1 Spring-Mass System Timestep Loop

```
// 计算每个质点所受的外力（这里只有重力）
for each Particle p : particles do
    p.frc = 0; // frc为质点所受力
    p.frc+ = p.mass * gravity;
end for
// 计算每个质点所受的内力（弹力和阻尼）从而求得质点所受合力
for all each Spring s(i, j) : springs do
    d = j.pos − i.pos; // 计算弹簧方向
    l = d.norm; // 计算弹簧长度
    v = j.vel − i.vel;
    f = (k_s(l/l_0 − 1) + k_d(v/l_0 · d/l)) d/l // 计算质点所受内力，其中l_0为弹簧原长
    i.frc+ = f;
    k.frc− = f;
end for
// 欧拉半隐式求解
for each Particle p : particles do
    p.vel+ = dt * (p.frc/p.mass);
    p.pos+ = dt * p.vel;
end for
```

其中关于阻尼k_d的部分暂时隐去，经过测试，是否有阻尼对结果影响不大，且阻尼过大可能会引发结果发散。

半隐式的方法缺点是不稳定，结果可能不收敛，而且波动比较大，结合隐式方法的思想，我对它做了一些改进：对x_{n+1}进行迭代来修正。

第一步：

$$x_{n+1} = x_n + hv_n$$

第二步：

$$v_{n+1} = v_n + hM^{-1}f(x_{n+1})$$

与半隐式不同，这里使用了由v_n粗略估计出来的x_{n+1}来计算力

第三步：

$$x_{n+1} = x_n + hv_{n+1}$$

即由v_{n+1}对x_{n+1}进行更新，并以此回到第二步迭代。

发现还是存在波动和不收敛的问题，但相对普通半隐式好了不少。

欧拉隐式方法：

对于基本公式，消去速度之后可以得到

$$y = x_n + hv_n + h^2 M^{-1} f_{ect}(x_n)$$

$$M(x-y) = h^2 f_{int}(x)$$

只需要求解方程得到x即可。

我们采用牛顿法迭代求解上述方程

$$x^{(0)} = y$$

$$x^{(k+1)} = x^{(k)} - (\nabla g(x^{(k)}))^{-1} g(x^{(k)})$$

$$\nabla g\left(x^{(k)}\right) = M - h^2 \nabla f_{int}(x)$$

在计算$\nabla f_{int}(x)$的时候，对于连接x_1，x_2的弹簧，

$$\frac{\partial f_1}{\partial x_1} = k\left(\frac{l}{\|r\|} - 1\right) I - kl\|r\|^{-3}$$

$$\frac{\partial f_1}{\partial x_2} = -\frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_1} = -\frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_2} = \frac{\partial f_1}{\partial x_1}$$

对每一个弹簧循环就行，这里可以把一个 3*3 的矩阵的赋值封装成函数，减少代码重复量。

对于边界点，我选择每次迭代后将 x 固定，当然设置力为 0 可能更加稳妥，有待改进。

欧拉隐式加速方法：

我们发现，对于$M(x-y)=\ h^2 f_{int}(x)$的解，我们可以将其化为最小化能量问题

$$x_{n+1}=arg\min_{x}\frac{1}{2}(x-y)^T M(x-y)+h^2E(x)$$

对于弹性势能 E，转化成最小化问题：

$$\frac{1}{2}(\|p_1-p_2\|-r)^2=\min\frac{1}{2}\|p_1-p_2-d\|$$

其中$d^T=(d_1^T,d_2^T\ldots d_s^T),d_i^T\in R^{1\times3},\|d_i\|=\ l_i,l_i$是第 i 个弹簧原长。

所以我们将原问题转化成了

$$x_{n+1}=arg\min_{x,\ d}\frac{1}{2}x^T(M+h^2L)x-h^2x^TJd-x^TMy$$

The matrices $\mathbf{L}\in\mathbb{R}^{3m\times3m}$, $\mathbf{J}\in\mathbb{R}^{3m\times3s}$ are defined as follows:

$$\mathbf{L}=\left(\sum_{i=1}^{s}k_i\mathbf{A}_i\mathbf{A}_i^{\mathsf{T}}\right)\otimes\mathbf{I}_3,\ \mathbf{J}=\left(\sum_{i=1}^{s}k_i\mathbf{A}_i\mathbf{S}_i^{\mathsf{T}}\right)\otimes\mathbf{I}_3$$

where $\mathbf{A}_i\in\mathbb{R}^m$ is the incidence vector of i-th spring, i.e., $A_{i,i_1}=1, A_{i,i_2}=-1$, and zero otherwise. Similarly, $\mathbf{S}_i\in\mathbb{R}^s$ is the i-th spring indicator, i.e., $S_{i,j}=\delta_{i,j}$. The matrix $\mathbf{I}_3\in\mathbb{R}^{3\times3}$ is the identity matrix and $\otimes$ denotes Kronecker product. Note that the matrix $\mathbf{L}$ is nothing but a stiffness-weighted Laplacian of the mass-spring system graph.

求导，立得

$$(M+h^2L)\,x=\ h^2Jd+My$$

采用 global/local 方法来解决

第一步：根据初始的 x（x=y）求出 d，由下式

$$d_i=\ l_i\frac{p_{i1}-p_{i2}}{\|p_{i1}-p_{i2}\|}$$

第二步：根据 d 解出新的 x。

注意：这里的矩阵都是在开始模拟之前就已经定好的，所以可以在模拟一开始（之所以不在 init 中是因为一开始 load 物体的时候会调用 init，这会让不需要模拟的物体加载的极慢，特别是点多的物体，所以放到了模拟开始）对矩阵进行初始化并预分解，加快效率。

第三步：迭代。

五.难点难题

1.发散问题：

在几个数据之间找平衡花了我一段时间，半隐式和隐式在步长为给定的 0.03 时，如果弹簧的弹性系数过大，会导致一次模拟跨度太大，导致结果发散，或者分母太小导致整体太大越界。解决方案是减小重力加速度来让每一帧更新的更慢，或者减小步长（同时还要控制帧的更新速率，否则会因为更新的太快而发生卡顿），但会明显让模拟变慢，对于隐式，甚至每一帧需要 2-3s，这是我们不能接受的，但加速算法可以很好的解决这个问题。我还对半隐式做了一些改变，见上功能实现。

2.初始化矩阵问题

发现对于 J 和 L,其实具有二次型的形式，将 ki 作为对角元素产生对角阵 K，并且将列向量排列成矩阵 A，就可以将 L 用

$$L = AKA^T$$

来表示，但在实际应用中，两次矩阵乘法会让初始化变得很慢，相比

之下对于边循环，赋值虽然写起来麻烦，但效率很高，所以最后采用了后者。

3.稀疏矩阵和稠密矩阵的转化

因为要解方程，所以不能使用 MatrixXd 类型，而是需要稀疏矩阵。试过“=”会报错。一开始想用三元组，发现写起来比较麻烦，查找资料发现可以使用 sparseView()函数。

4.矩阵赋值问题

发现每一次计算都是一个 3*3 的矩阵，想找个库函数能直接把一个 3*3 的矩阵加入但没有成功，后来想到可以传 MatrixXd*指针，另写一个函数对九宫格赋值。

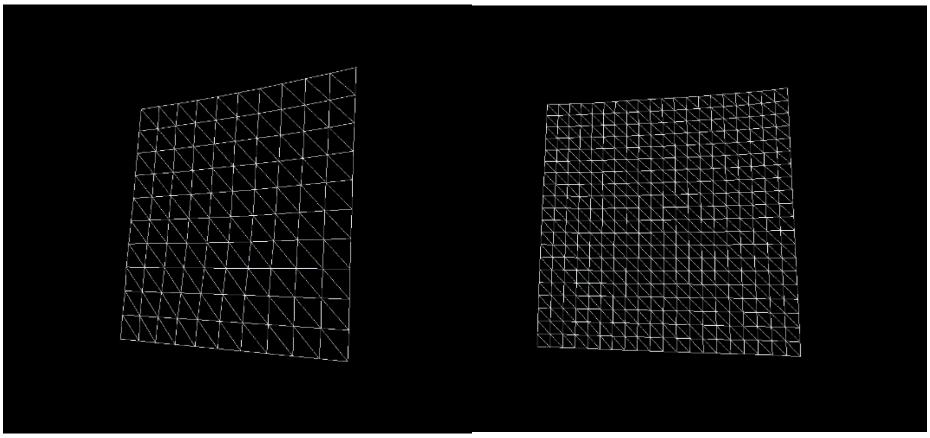
5.类继承问题

主要问题是在调用 Run 的时候，是用 MassSpring 中的 simulate 调用的，但 MassSpring 不是按右侧按钮产生的，是通过上方按钮开始模拟，但是我没找到上方按钮的代码位置，所以还是写成了三个函数。

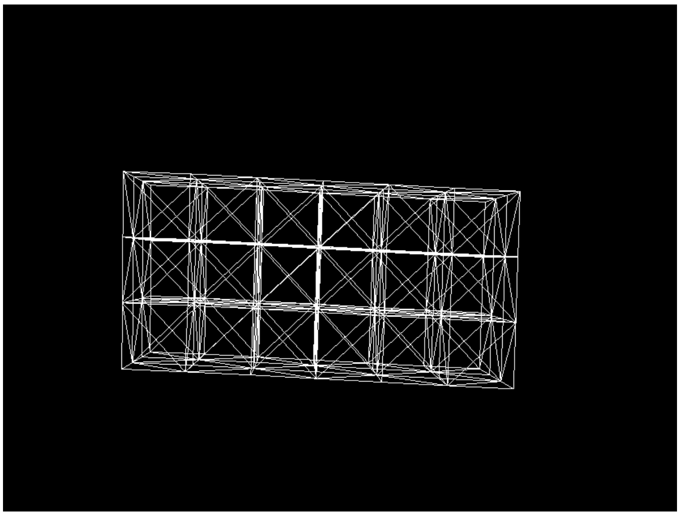
六.实验结果

视频在文件夹中，图片为收敛结果（或者收敛过程太长，截了一张）。

半隐式方法：

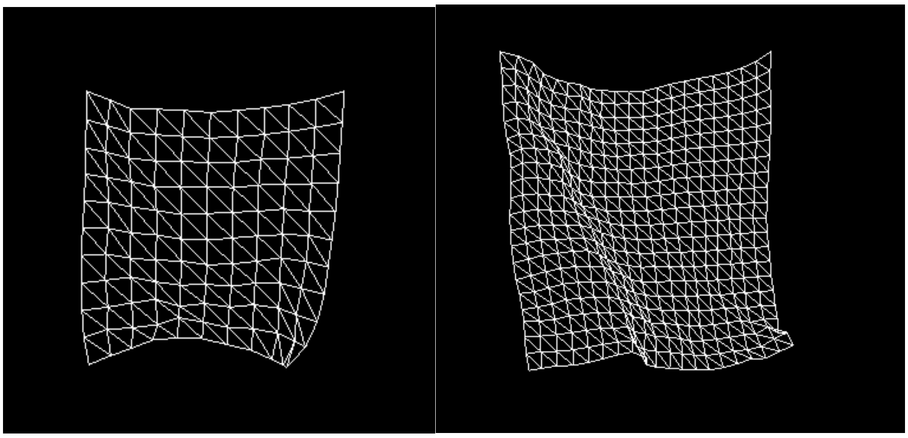


Ks = 1000

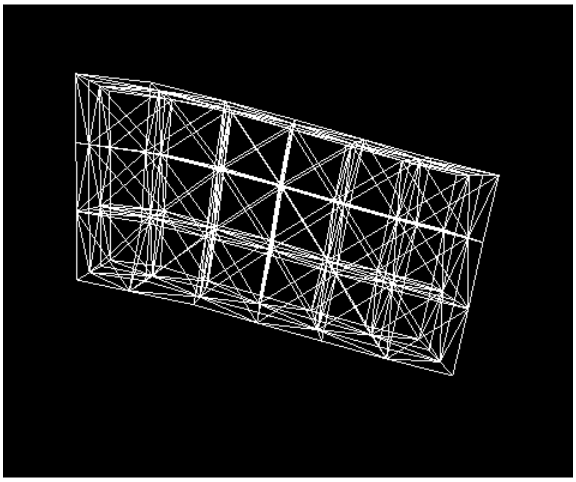


Ks = 100

隐式方法：

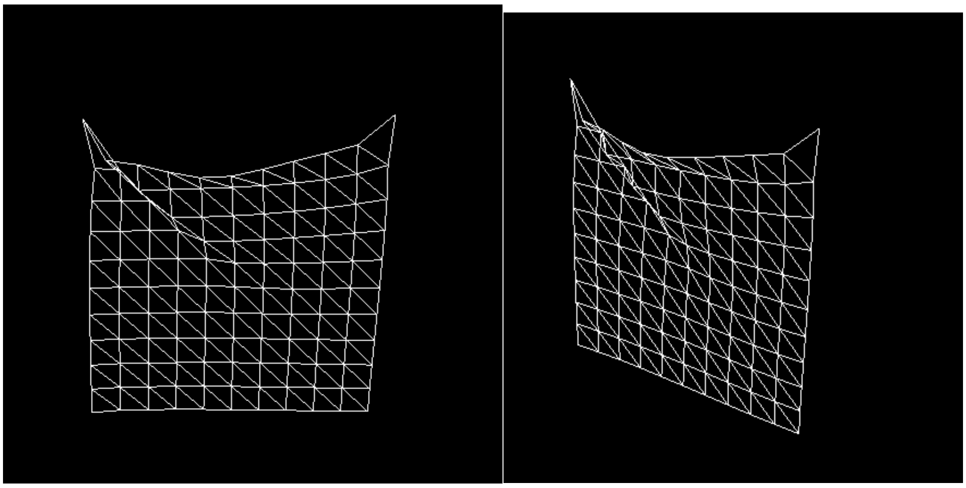


Ks =1000

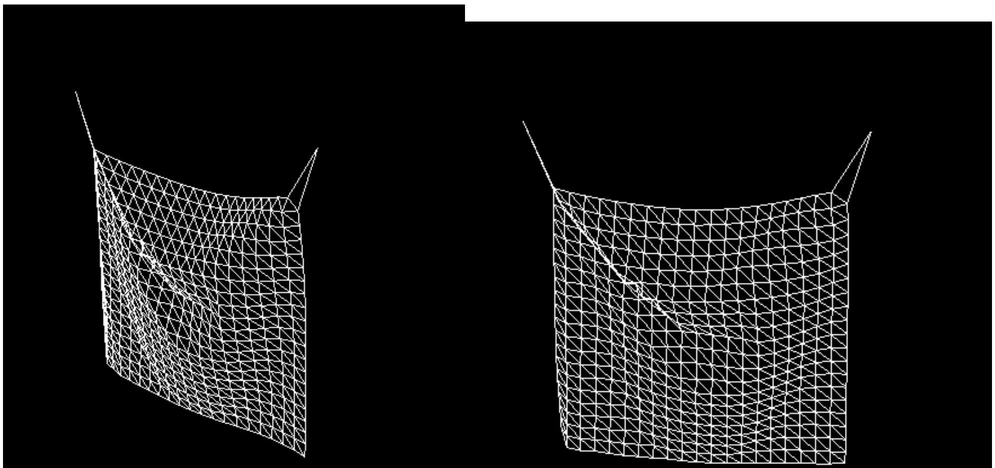


Ks = 1000

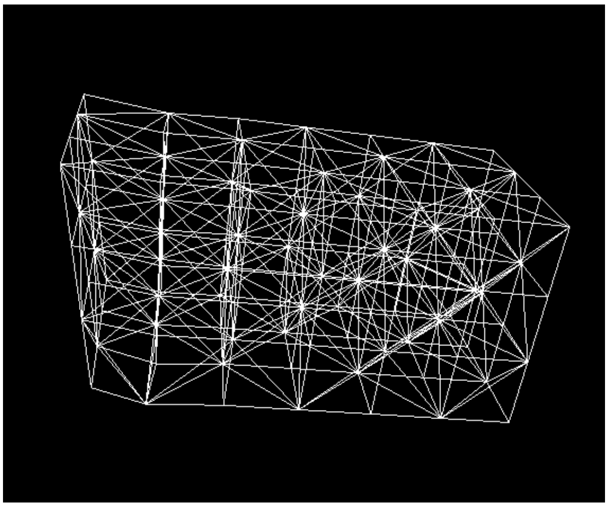
隐式方法加速：



Ks =1000， 收敛时间 6 ‘20



Ks =100000， 收敛时间 5 ‘20



Ks =1000

总的来说，加速隐式方法在各个方面都比前两种方法好得多，无论是真实性，收敛速度，抗发散，运行速度都最优，唯一有点小瑕疵的是对于 dense 这种点多的情形，会导致两个固定点拉的太长，可以考虑根据密度变换质量。隐式算法运行时间过长，半隐式波动又过于明显。值得一提的是，相同的弹性系数，半隐式表现出来的比其真正的系数要大很多，可以发现基本不发生形变，值得探究为什么。

七．问题与展望

7.1　遇到的问题

- 类的继承
- 隐式算法速度慢
- 半隐式算法易发散（设置阀值？）

7.2 future work

- 重量动态改变
- 将三个方法编入三个类里面
- 对于固定点的更好操作，力的平衡
- 处理碰撞
- 更快的收敛速度
- 添加阻尼，空气阻力