Homework 8 作业报告 by 76-朱雨田

本次作业要求实现半隐式欧拉方法、隐式欧拉方法与 Liu13 三种弹簧质点模型算法，并设计基于惩罚力的物理球体碰撞模拟。

因为本次作业的展示图多为 gif，因此报告中不附图，仅引用压缩包内附带的 gif。

因为这次的原理推导有不少在我的能力范围之外，而且公式都给出来了不需要自己推（？）所以这次的代码实现并没有过多地思考原理方面的东西。因为其他科目也比较忙，所以隐式欧拉和 Liu13 算法的代码基本上都是参考自己一周目的代码编写的。这次报告中不会过多地讲解数学、物理方面的原理推导，而是更侧重于效果的分析与数据对比。

附件说明

文件	说明
./gif/1.gif	20x20 网格，半隐式欧拉方法，h=0.001, stiffness=18.00, damping=0.200 加速后的结果
./gif/2.gif	20x20 网格，隐式欧拉方法，h=0.02, stiffness=18.00，无 damping，贴有纹理
./gif/3.gif	20x20 网格，隐式欧拉方法，h=0.02，stiffness=180.00，无 damping，贴有纹理
./gif/4.gif	20x20 网格，Liu 13 方法，h=0.02，stiffness=1800.00，无 damping，贴有纹理
./gif/5.gif	20x20 网格，Liu 13 方法，h=0.02，stiffness=180.00，无 damping，有球体

实现细节

半隐式欧拉方法

对于 damping，我并没有像文档一样在每步都乘以 damping，而是在每步执行 `vel *= pow(damping, h)`，并将 damping 设置为一个相对较小的值。这样可以保证 h 在不同取值下，同一段物理时间下的速度衰减效果相近。

早期编写的时候因为一些数据问题，所以会产生与预期不同的爆炸。在此期间我用了许多方法进行 Debug：

- **输出能量。**在每步完成后可以分别计算系统的动能、弹性势能、重力势能，并将其求和以验证能量的变化趋势与守恒性质。

```
const bool calculate_energy = true;
if (calculate_energy)
{
    double energy1 = 0, energy2 = 0, energy3 = 0;
    for (int i = 0; i < n_vertices; i++)
    {
        energy1 += 0.5 * mass_per_vertex * vel.row(i).squaredNorm();
        energy2 += -mass_per_vertex * gravity.dot(X.row(i));
    }
    int i = 0;
```

```
    for (const auto& e : E)
    {
        energy3 += 0.5 * stiffness * pow((X.row(e.first) -
X.row(e.second)).norm() - E_rest_length[i], 2);
        i++;
    }
    std::cout << step_n << ": " << energy1 + energy2 + energy3 << " = \t" <<
energy1 << " + \t" << energy2 << " + \t" << energy3 << std::endl;
}
```

在正确编写后，这段代码的输出结果大致如下：

```
1: -4.7802e-05 =        4.78023e-05 +   -9.56045e-05 +  1.86481e-10
2: -9.56041e-05 =       0.000191209 +   -0.000286813 +  1.0148e-10
3: -0.000143406 =       0.00043022 +    -0.000573627 +  1.58205e-10
4: -0.000191209 =       0.000764836 +   -0.000956045 +  2.31925e-10
5: -0.000239011 =       0.00119506 +    -0.00143407 +   2.07203e-10
6: -0.000286813 =       0.00172088 +    -0.00200769 +   1.57373e-10
7: -0.000334615 =       0.00234231 +    -0.00267692 +   2.05513e-10
8: -0.000382417 =       0.00305934 +    -0.00344176 +   3.37252e-10
9: -0.00043022 =        0.00387198 +    -0.0043022 +    4.89557e-10
10: -0.000478022 =      0.00478022 +    -0.00525825 +   7.5085e-10
11: -0.000525824 =      0.00578407 +    -0.00630989 +   1.33713e-09
12: -0.000573626 =      0.00688352 +    -0.00745715 +   2.42527e-09
13: -0.000621427 =      0.00807857 +    -0.0087 +       4.17909e-09
14: -0.000669229 =      0.00936923 +    -0.0100385 +    6.96226e-09
15: -0.00071703 =       0.0107555 +     -0.0114725 +    1.14106e-08
16: -0.000764831 =      0.0122374 +     -0.0130022 +    1.8299e-08
17: -0.000812631 =      0.0138148 +     -0.0146275 +    2.85164e-08
18: -0.00086043 =       0.0154879 +     -0.0163484 +    4.32903e-08
19: -0.000908228 =      0.0172565 +     -0.0181648 +    6.43864e-08
20: -0.000956025 =      0.0191208 +     -0.0200769 +    9.40691e-08
...
```

其中右式三项分别为动能、重力势能、弹性势能。总能量的不断减小，是 damping 作用于动能的结果。

- **逐帧模拟。**这次的框架虽然没有自带逐帧模拟的功能，但我们可以在 `step()` 函数的末尾加入一个 `std::cin >> a;` 以暂停程序的运行。这样可以非常方便地进行逐帧测试。

隐式欧拉算法

固定点约束

对于固定点的约束，我这里采用了选择矩阵的方法。

具体而言，选择矩阵是一个 $3m \times 3n$ 的矩阵，其中 n 为顶点个数，m 为非固定顶点的个数。第 i 行对应第 i 个非固定顶点，矩阵中其编号对应列的值为 1，其他列对应的值均为 0。将这样的矩阵构造后，可以参与进运算中，以实现固定顶点的效果。选择矩阵代入方程的形式可以参考 2020 年计算机图形学课程的文档：

USTC_CG/Homeworks/6_MassSpring/documents/README.md at master · Ubpa/USTC_CG (github.com)

选择矩阵可以在构造函数中就初始化。我构造选择矩阵的代码如下：

```cpp
// Select Matrix (K & Kt)
std::vector<Triplet<double>> tripletList;
tripletList.reserve(n_active);

K = SparseMatrix<double>(3 * n_active, 3 * n_vertices);
Kt = SparseMatrix<double>(3 * n_vertices, 3 * n_active);
int active_i = 0;
for (int i = 0; i < n_vertices; i++)
{
    if (!dirichlet_bc_mask[i])
    {
        tripletList.push_back({ active_i * 3 + 0, i * 3 + 0, 1 });
        tripletList.push_back({ active_i * 3 + 1, i * 3 + 1, 1 });
        tripletList.push_back({ active_i * 3 + 2, i * 3 + 2, 1 });
        active_i++;
    }
}
K.setFromTriplets(tripletList.begin(), tripletList.end());
Kt = K.transpose();
```

多次迭代

除此之外，我还实现了多次的迭代。预先设定循环终止条件常数 ϵ 值，在第二次牛顿迭代后检查 $\|x' - x\|^2$ 的值（其中 $x' \in \mathbb{R}^{3n\times 1}$ 为上一次迭代后的位置矢量，x 则为该次迭代后的位置矢量），若 $\|x' - x\|^2 < \epsilon$，则认为能量已经收敛，终止循环。

在这种模式下，我们还可以输出每一帧的迭代次数。可以发现当布料摆至低点时，迭代的次数会明显增加。这可能是因为布料速度较快，每步的位移较大，所以需要更多次的迭代来接近能量低点。

每步时长输出

因为每步的迭代次数会有不同，因此比较单步的时长是无法很好地比较不同条件下的效率差异的。这里我利用 `TIC TOC` 宏，进一步统计了不同步长下的平均用时和迭代次数。代码如下：

```cpp
// step 函数开始
TIC(step)
static int step_n = 0;
static double sum_step_time = 0;
static int sum_itr = 0;
step_n++;

// 隐式欧拉方法结束前
sum_itr += itr + 1;
std::cout << "Average Iteration Number: " << (double)sum_itr / step_n << std::endl;

// step 函数结束前
TOC(step)
double steptime = std::chrono::duration_cast<std::chrono::microseconds>(end_step -
start_step).count();
sum_step_time += steptime;
std::cout << "Average Step Time: " << sum_step_time / step_n << " microseconds.\n";
```

Hessian 矩阵正定性

因为 Hessian 矩阵无法保证正定的情况下无法保证牛顿迭代法的收敛，从而会造成爆炸，所以我们需要手动调整 Hessian 矩阵以使得矩阵正定。

我采用了文档中的第一种方法对 Hessian 矩阵进行调整。该方法较为简单、近似，但经过测试后能发现经过这种调整后的模拟效果仍然和半隐式欧拉方法的模拟效果相近。

```
Matrix3d he = k * (l / rl - 1) * Matrix3d::Identity() - k * l / rl / rl / rl * r * r.transpose();
if (l > rl) he = -k * l / rl / rl / rl * r * r.transpose();
```

Liu 13 方法

其实感觉只要有了公式，加速方法的代码写起来比隐式欧拉还顺手许多。其主要就只有预计算矩阵和 Local/Global 迭代两部分，而 Local/Global 迭代的编写比起 ARAP 要容易非常多。这里同样也采用了和隐式欧拉方法一样的迭代终止条件。

下面是迭代部分的主要代码。

```
// Local Phase
int i = 0;
for (const auto& e : E)
{
    Vector3d vec;
    for (int c = 0; c < 3; c++)
        vec(c) = x(e.first * 3 + c) - x(e.second * 3 + c);
    Vector3d di = E_rest_length[i] / vec.norm() * vec;
    for (int c = 0; c < 3; c++) d(i * 3 + c) = di(c);
    i++;
}

// Global Phase
xa = solver.solve(K * (h * h * J * d + M * y - (M + h * h * L) * b));
x = Kt * xa + b;
```

球体碰撞约束

```
Vector3d vec = X.row(i).transpose() - center;
double dist = vec.norm();
if (dist < collision_scale_factor * radius)
    force.row(i) = collision_penalty_k * (collision_scale_factor * radius - dist) * vec / dist;
```

同样只需要照着公式写即可。惩罚力的方法的原理是为顶点提供额外的外力，这种外力在三种弹簧质点模型算法中都是很方便引入的，只需在重力、风力项的基础上再加一项碰撞力即可。

数据对比研究

1. 不同算法运行效率（无球体）

下列数据均在 Release 编译选项下测得。

网格均为 grid20x20.usda。Stiffness 均为 18.00。保证不发生爆炸。

欧拉半隐式方法

参数	值
h	0.001
damping	0.200
平均每步时长 / μs	**27.6831**

欧拉隐式方法

参数	值
h	0.020
damping	1.000
循环终止条件	$\|\|\mathbf{x}' - \mathbf{x}\|\|^2 < 10^{-3}$
平均迭代次数	**3.29087**
平均每步时长 / μs	**11863.7**

Liu 13 方法

参数	值
h	0.020
damping	1.000
循环终止条件	$\|\|\mathbf{x}' - \mathbf{x}\|\|^2 < 10^{-3}$
平均迭代次数	**6.19664**
平均每步时长 / μs	**498.456**

2. 不同算法运行效率（有球体）

下列数据均在 Release 编译选项下测得。

网格均为 grid20x20.usda。Stiffness 均为 180.00. 球体位置为 $(0, 0, 0.5)$，半径为 0.4，伸缩系数为 1.1，$k^{\text{penalty}} = 8000$。

保证不发生爆炸。

欧拉半隐式方法

测量了整个时间轴，布料始终覆盖在球体上。

参数	值
h	0.001
damping	0.200
平均每步时长 / μs	**29.6346**

欧拉隐式方法

只取了测量时间在 100s 以内的数据，因为 100s 后布料滑落球体，不再与球体交互。

参数	值
h	0.020
damping	1.000
循环终止条件	$\|\|\mathbf{x}' - \mathbf{x}\|\|^2 < 10^{-3}$
平均迭代次数	**7.06966**
平均每步时长 / μs	**24222.8**

Liu 13 方法

测量了整个时间轴，布料始终覆盖在球体上。

参数	值
h	0.020
damping	1.000
循环终止条件	$\|\|\mathbf{x}' - \mathbf{x}\|\|^2 < 10^{-3}$
平均迭代次数	**38.9329**
平均每步时长 / μs	**2227.29**

可以看出，不管是哪种方法，加上球体都会导致

3. 不同终止条件下的迭代次数与每步时长比较

网格均为 grid20x20.usda。参数均为 h=0.020, stiffness=180.00，无球体。保证不发生爆炸。

欧拉隐式方法

终止条件为 1 时仍然能模拟得到很好的效果。

$\|\mathbf{x}' - \mathbf{x}\|^2$	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
平均迭代次数	2	2.00719	2.98561	3.71703	5	7.29496
平均每步时长 / μs	7244.6	7396.92	10210.5	13196.2	18122.2	25914.2

Liu 13 方法

终止条件为 1, 0.1 时布料边缘有明显的三角面翻折。

$\|\mathbf{x}' - \mathbf{x}\|^2$	1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
平均迭代次数	2	2	2.18465	9.09113	54.6866	100.892
平均每步时长 / μs	202.981	198.715	213.396	619.432	3015.32	5941.71

从以上数据也能看出 Liu 13 方法收敛速度快、残差下降慢的特点。