

# Lab 4 报告

蒋锦豪 2400011785

2025 年 12 月 4 日

由于本 Lab 的效果多为动图，无法放进报告中，故效果图数量较少，以运行效果为准。

## Task 1: Inverse Kinematics

### Sub-Task 1

前向运动学的原理如下：节点  $P_0, \dots, P_n$  的初始位置记作  $\mathbf{p}_0^{\text{st}}, \dots, \mathbf{p}_n^{\text{st}}$ ，向下一个关节的位移即为  $\mathbf{l}_i = \mathbf{p}_i^{\text{st}} - \mathbf{p}_{i-1}^{\text{st}} (i = 1, \dots, n)$ ，每个节点上都带有一个局部坐标系  $C_i$ 。对除去  $P_n$  外的每个关节  $P_j$  指定一个旋转矩阵  $\mathbf{R}_j^{\text{loc}} (j = 1, \dots, n-1)$ ，每个关节  $P_j$  处的旋转带动其后的所有关节和骨骼旋转。因此，节点  $P_j$  的总的旋转矩阵  $\mathbf{R}_j^{\text{tot}}$  即为它的局部坐标系  $C_j$  相对于根节点的坐标系  $C_0$  的旋转与它本身的旋转的复合。而  $C_j$  的旋转事实上就是由  $P_{j-1}$  节点的旋转定义的，因此  $\mathbf{R}_j^{\text{tot}} = \mathbf{R}_{j-1}^{\text{tot}} \mathbf{R}_j^{\text{loc}}$ 。

现在就可以从根节点开始计算每个子节点的位置  $\mathbf{p}_j^{\text{ed}} (j = 1, \dots, n)$ 。根节点的位置  $\mathbf{p}_0^{\text{ed}} = \mathbf{p}_0^{\text{st}}$  已知，而每个节点的位置可以通过其父节点的位置加上旋转后的位移向量得到，即

$$\mathbf{p}_j^{\text{ed}} = \mathbf{p}_{j-1}^{\text{ed}} + \mathbf{R}_{j-1}^{\text{tot}} \mathbf{l}_j$$

代码实现如下：

```
1 void ForwardKinematics(IKSystem & ik, int StartIndex) {
2     if (StartIndex == 0) {
3         ik.JointGlobalRotation[0] = ik.JointLocalRotation[0];
4         ik.JointGlobalPosition[0] = ik.JointLocalOffset[0];
5         StartIndex             = 1;
6     }
7     for (int i = StartIndex; i < ik.JointLocalOffset.size(); i++) {
8         // your code here: forward kinematics, update JointGlobalPosition and JointGlobalRotation
9         ik.JointGlobalRotation[i] = ik.JointGlobalRotation[i - 1] * ik.JointLocalRotation[i];
10        ik.JointGlobalPosition[i] = ik.JointGlobalPosition[i - 1] + ik.JointGlobalRotation[i - 1] *
11            ik.JointLocalOffset[i];
12    }
```

12 }

## Sub-Task 2

CCD IK 的思想是从末端关节开始, 依次调整每个关节的旋转, 使得末端关节逐渐接近目标位置.

具体地, 假定当前铰接刚体系统共有  $n + 1$  个关节与  $n$  个连接关节的骨骼, 希望末端的关节位置  $\mathbf{p}_n$  距离目标  $t$  尽可能接近. 于是遍历第  $n - 1, \dots, 0$  个关节, 每次求解关节  $i$  处的旋转矩阵  $\mathbf{R}_i^{\text{loc}}$  使得调整后  $\mathbf{p}_i, \mathbf{p}_n$  和  $t$  三点共线 (在 `glm` 库中, 求解将向量  $\mathbf{a}$  旋转到向量  $\mathbf{b}$  对应的四元数可以用 `glm::rotate` 函数实现.), 然后将更新后的  $\mathbf{R}_i^{\text{loc}}$  应用于第  $i$  个关节及其后续的所有关节, 更新它们的位置. 重复上述过程直到末端关节  $\mathbf{p}_n$  足够接近目标  $t$  或者达到最大迭代次数为止. 代码实现如下:

```

1 void InverseKinematicsCCD(IKSystem & ik, const glm::vec3 & EndPosition, int maxCCDIKIteration, float
2   eps) {
3   ForwardKinematics(ik, 0);
4   // These functions will be useful: glm::normalize, glm::rotation, glm::quat * glm::quat
5   for (int CCDIKIteration = 0; CCDIKIteration < maxCCDIKIteration && glm::l2Norm(ik.
6     EndEffectorPosition() - EndPosition) > eps; CCDIKIteration++) {
7     // your code here: ccd ik
8     for (int i = ik.NumJoints() - 2; i >= 0; i--) {
9       glm::vec3 v_st = ik.JointGlobalPosition[i], v_ed = ik.JointGlobalPosition[ik.
10      JointLocalOffset.size() - 1];
11      glm::vec3 v = glm::normalize(v_ed - v_st), u = glm::normalize(EndPosition - v_st);
12      glm::quat q = glm::rotation(v, u);
13      ik.JointLocalRotation[i] = q * ik.JointLocalRotation[i];
14      ForwardKinematics(ik, 0);
15    }
16  }
17 }
```

## Sub-Task 3

FABR IK 的思想是通过两次遍历关节链, 根据骨骼长度调整关节位置, 使得末端关节接近目标位置.

具体地, 假定当前铰接刚体系统共有  $n + 1$  个关节与  $n$  个连接关节的骨骼, 希望末端的关节位置  $\mathbf{p}_n$  距离目标  $t$  尽可能接近.

首先是 Backward 步骤的计算. 将末端关节  $\mathbf{p}_n$  移动到目标位置  $t$ , 记  $\mathbf{p}_n^{\text{back}} = t$ . 接着, 从末端关节向根节点遍历每个关节  $i(i = n - 1, \dots, 0)$ , 将  $\mathbf{p}_i$  更新到  $\mathbf{p}_i^{\text{back}}$  使得

$$|\mathbf{p}_{i+1}^{\text{back}} - \mathbf{p}_i^{\text{back}}| = |l_{i+1}|, \quad \mathbf{p}_i, \mathbf{p}_i^{\text{back}}, \mathbf{p}_{i+1}^{\text{back}} \text{ 共线}$$

于是可得

$$\mathbf{p}_i^{\text{back}} = \mathbf{p}_{i+1}^{\text{back}} + \frac{\mathbf{p}_i - \mathbf{p}_{i+1}^{\text{back}}}{|\mathbf{p}_i - \mathbf{p}_{i+1}^{\text{back}}|} |\mathbf{l}_{i+1}|$$

如此往复直到根节点位置. 现在得到了一组新的关节位置  $\{\mathbf{p}_i^{\text{back}}\}_{i=1}^n$ .

接下来是 Forward 步骤的计算. 记  $\mathbf{p}_0^{\text{for}} = \mathbf{p}_0^{\text{st}}$ . 接着从根节点向末端遍历每个关节  $i(i = 1, \dots, n)$ , 将  $\mathbf{p}_i^{\text{back}}$  更新到  $\mathbf{p}_i^{\text{for}}$  使得

$$|\mathbf{p}_i^{\text{for}} - \mathbf{p}_{i-1}^{\text{for}}| = |\mathbf{l}_i|, \quad \mathbf{p}_i^{\text{back}}, \mathbf{p}_i^{\text{for}}, \mathbf{p}_{i-1}^{\text{for}} \text{ 共线}$$

于是可得

$$\mathbf{p}_i^{\text{for}} = \mathbf{p}_{i-1}^{\text{for}} + \frac{\mathbf{p}_i^{\text{back}} - \mathbf{p}_{i-1}^{\text{for}}}{|\mathbf{p}_i^{\text{back}} - \mathbf{p}_{i-1}^{\text{for}}|} |\mathbf{l}_i|$$

如此往复直到末端节点位置. 现在得到了一组新的关节位置  $\{\mathbf{p}_i^{\text{for}}\}_{i=1}^n$ . 这组新的关节满足骨骼长度约束, 并且  $\mathbf{p}_n$  相比之前更接近目标  $\mathbf{t}$ .

重复上述的 Backward 和 Forward 步骤直到终止条件为止. 代码实现如下:

```

1 void InverseKinematicsFABR(IKSystem & ik, const glm::vec3 & EndPosition, int maxFABRIKIteration,
2     float eps) {
3     ForwardKinematics(ik, 0);
4     int nJoints = ik.NumJoints();
5     std::vector<glm::vec3> backward_positions(nJoints, glm::vec3(0, 0, 0)), forward_positions(
6         nJoints, glm::vec3(0, 0, 0));
7     for (int IKIteration = 0; IKIteration < maxFABRIKIteration && glm::l2Norm(ik.EndEffectorPosition(
8         ) - EndPosition) > eps; IKIteration++) {
9         // task: fabr ik
10        // backward update
11        glm::vec3 next_position      = EndPosition;
12        backward_positions[nJoints - 1] = EndPosition;
13
14        for (int i = nJoints - 2; i >= 0; i--) {
15            // your code here
16            glm::vec3 dir          = glm::normalize(ik.JointGlobalPosition[i] - backward_positions[i +
17                1]);
18            float      len          = glm::length(ik.JointOffsetLength[i + 1]);
19            backward_positions[i] = backward_positions[i + 1] + len * dir;
20        }
21
22        // forward update
23        glm::vec3 now_position = ik.JointGlobalPosition[0];
24        forward_positions[0] = ik.JointGlobalPosition[0];
25        for (int i = 0; i < nJoints - 1; i++) {
26            // your code here
27        }
28    }
29}
```

```

23         glm::vec3 dir          = glm::normalize(backward_positions[i + 1] - forward_positions[
24             i]);
25         float len            = glm::length(ik.JointOffsetLength[i + 1]);
26         forward_positions[i + 1] = forward_positions[i] + len * dir;
27     }
28     ik.JointGlobalPosition = forward_positions; // copy forward positions to joint_positions
29 }
30
31 // Compute joint rotation by position here.
32 for (int i = 0; i < nJoints - 1; i++) {
33     ik.JointGlobalRotation[i] = glm::rotation(glm::normalize(ik.JointLocalOffset[i + 1]), glm::
34         normalize(ik.JointGlobalPosition[i + 1] - ik.JointGlobalPosition[i]));
35 }
36 ik.JointLocalRotation[0] = ik.JointGlobalRotation[0];
37 for (int i = 1; i < nJoints - 1; i++) {
38     ik.JointLocalRotation[i] = glm::inverse(ik.JointGlobalRotation[i - 1]) * ik.
39     JointGlobalRotation[i];
}
ForwardKinematics(ik, 0);
}

```

## Sub-Task 4

使用简单的直线和三角函数绘制了姓名首字母, 绘图的参数化实现 `custom_x` 和 `custom_y` 如下所示:

```

1 float pi = glm::pi<float>();
2 float clamp(float t, float min_t, float max_t) {
3     return (t < max_t && t >= min_t) ? 1.0f : 0.0f;
4 }
5 float custom_x(float t) {
6     return clamp(t, 0, pi) * (-4.0f + t * 2 / pi)
7     + clamp(t, pi, 2 * pi) * (-3.0f)
8     + clamp(t, 2 * pi, 3 * pi) * (0.5f * cos(t - 2 * pi) - 3.5f)
9     + clamp(t, 3 * pi, 4 * pi) * (-1.0f + (t - 3 * pi) * 2 / pi)
10    + clamp(t, 4 * pi, 5 * pi) * 0.0f
11    + clamp(t, 5 * pi, 6 * pi) * (0.5f * cos(t - 5 * pi) - 0.5f)
12    + clamp(t, 6 * pi, 7 * pi) * 2.0f
13    + clamp(t, 7 * pi, 8 * pi) * (2.0f + (t - 7 * pi) * 2 / pi)
14    + clamp(t, 8 * pi, 9 * pi) * 4.0f;
15 }

```

```

16 float custom_y(float t) {
17     return clamp(t, 0, pi) * 1.0f
18     + clamp(t, pi, 2 * pi) * (1.0f - (t - pi) * 1.5 / pi)
19     + clamp(t, 2 * pi, 3 * pi) * (0.5f * (-sin(t - 2 * pi)) - 0.5f)
20     + clamp(t, 3 * pi, 4 * pi) * 1.0f
21     + clamp(t, 4 * pi, 5 * pi) * (1.0f - (t - 4 * pi) * 1.5 / pi)
22     + clamp(t, 5 * pi, 6 * pi) * (0.5f * (-sin(t - 5 * pi)) - 0.5f)
23     + clamp(t, 6 * pi, 7 * pi) * (1.0f - (t - 6 * pi) * 2 / pi)
24     + clamp(t, 7 * pi, 8 * pi) * 0.0f
25     + clamp(t, 8 * pi, 9 * pi) * (1.0f - (t - 8 * pi) * 2 / pi);
26 }

```

此外在 `tasks.cpp` 文件中的 `BuildCustomTargetPosition()` 函数也进行了对参数 `t` 的范围和偏移量的适当修改以适应新的图形. 最终实现的效果图如下:

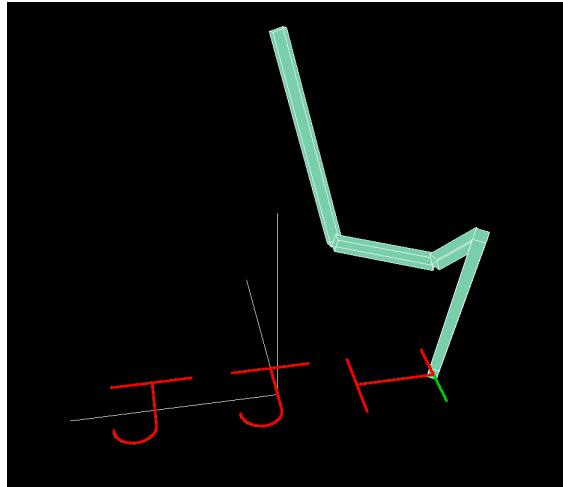


图 1: 绘制姓名首字母的效果图

## 问题回答

1. 如果目标位置太远, 无法到达, IK 结果会怎样?

答: 根据 IK 的原理,(在没有角度限制的情况下) 各个节点将连成指向目标位置的直线, 也即能使得末端节点最接近目标位置的关节旋转方式.

2. 比较 CCD IK 和 FABR IK 所需要的迭代次数.

答: FABR IK 的迭代次数更少. CCD IK 的更新方式是旋转关节, 每次只能逐关节地改善末端位置, 且改变关节也会对之前的迭代结果产生影响; 而 FABR IK 的更新方式是直接移动关节点坐标, 是一种更加整体且直接的方式, 需要的迭代次数更少.

3. 由于 IK 是多解问题, 在个别情况下, 会出现前后两帧关节旋转抖动的情况. 怎样避免或是缓解这种情况?

答：一种方法是对计算出的多个解优先选择与上一帧最接近的解，但是会产生额外的性能开销；另一种方法是限制每两帧之间各个关节的角度变化，这在物理上更加合理，但是可能产生延迟。

## Task 2: Mass-Spring System

隐式欧拉积分需要求解的方程如下：

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \mathbf{v}(t_{k+1})\Delta t$$

$$\mathbf{v}(t_{k+1}) = \mathbf{v}(t_k) + \mathbf{M}^{-1}\mathbf{f}(t_{k+1})\Delta t$$

这样需要解一个隐式方程组。将  $\mathbf{v}(t_{k+1})$  代入  $\mathbf{x}(t_{k+1})$  中可得

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + (\mathbf{v}(t_k) + \mathbf{M}^{-1}\mathbf{f}(t_{k+1})\Delta t)\Delta t$$

为了方便考虑，把力  $\mathbf{f}$  分成内力和外力两部分：

$$\mathbf{f}(t_{k+1}) = \mathbf{f}_{\text{int}}(t_{k+1}) + \mathbf{f}_{\text{ext}}$$

其中  $\mathbf{f}_{\text{ext}}$  作为外力与时间和质点的位置无关。于是进一步整理可得

$$\mathbf{x}(t_{k+1}) = [\mathbf{x}(t_k) + \Delta t\mathbf{v}(t_k) + (\Delta t)^2\mathbf{M}^{-1}\mathbf{f}_{\text{ext}}] + (\Delta t)^2\mathbf{M}^{-1}\mathbf{f}_{\text{int}}(t_{k+1})$$

等号右边前半部分全部为已知量，不妨记作  $\mathbf{y}(t_k)$ 。于是方程组即为

$$\mathbf{x}(t_{k+1}) - \mathbf{y}(t_k) - (\Delta t)^2\mathbf{M}^{-1}\mathbf{f}_{\text{int}}(t_{k+1}) = \mathbf{0}$$

接下来需要将内力  $\mathbf{f}_{\text{int}}(t_{k+1})$  表示成位置  $\mathbf{x}(t_{k+1})$  的函数。对于质点  $i$  与  $j$  之间的弹簧，其弹性势能

$$E_{ij} = \frac{1}{2}k_{ij}(\|\mathbf{x}_j - \mathbf{x}_i\| - l_{ij})^2$$

于是

$$\mathbf{f}_{ij} = -\frac{\partial E_{ij}}{\partial \mathbf{x}_i}$$

从而对于质点  $i$ ，其内力  $\mathbf{f}_{i,\text{int}}$  就可以写作

$$\mathbf{f}_{i,\text{int}} = -\sum_{j \in \Omega(i)} \frac{\partial E_{ij}}{\partial \mathbf{x}_i} = -\frac{\partial E}{\partial \mathbf{x}_i}$$

其中  $E$  为弹簧系统的总能量（与  $i$  不相连的弹簧的势能对  $\mathbf{x}_i$  的梯度为  $\mathbf{0}$ ，因此可以并入求和项中），其自变量为各质点的位置向量  $\mathbf{x}$ 。将所有质点的内力堆叠起来，我们就有

$$\mathbf{x}(t_{k+1}) - \mathbf{y}(t_k) + (\Delta t)^2\mathbf{M}^{-1}\frac{\partial E(\mathbf{x}(t_{k+1}))}{\partial \mathbf{x}} = \mathbf{0}$$

于是，可以构造辅助函数

$$g(\mathbf{x}) = \frac{1}{2(\Delta t)^2}(\mathbf{x} - \mathbf{y}(t_k))^t \mathbf{M}(\mathbf{x} - \mathbf{y}(t_k)) + E(\mathbf{x})$$

则有

$$\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} = \frac{\mathbf{M}}{(\Delta t)^2} \left[ \mathbf{x} - \mathbf{y}(t_k) + (\Delta t)^2 \mathbf{M}^{-1} \frac{\partial E(\mathbf{x})}{\partial \mathbf{x}} \right]$$

可以发现当  $\mathbf{x} = \mathbf{x}(t_{k+1})$  时恰好有  $\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{0}$ . 于是, 目标就转化为最小化问题

$$\mathbf{x}(t_{k+1}) = \arg \min_{\mathbf{x}} g(\mathbf{x})$$

采取牛顿法解决上述问题. 牛顿法的基本思想是在每轮迭代中使用一个二次函数逼近目标函数, 然后求解该二次函数的极小值对应的  $\mathbf{x}$  作为下一次迭代的点. 迭代过程每次通过尝试解  $\mathbf{x}^{(i)}$  计算下一次的解  $\mathbf{x}^{(i+1)}$ . 对  $g(\mathbf{x})$  在  $\mathbf{x}^{(i)}$  处进行二阶泰勒展开, 有

$$g(\mathbf{x}) = g(\mathbf{x}^{(i)}) + \nabla g(\mathbf{x}^{(i)}) \cdot (\mathbf{x} - \mathbf{x}^{(i)}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(i)})^T \mathbf{H}_g(\mathbf{x}^{(i)}) (\mathbf{x} - \mathbf{x}^{(i)}) + o(||\mathbf{x} - \mathbf{x}^{(i)}||^3)$$

其中  $\nabla = \frac{\partial}{\partial \mathbf{x}}$ ,  $\mathbf{H}_g$  为  $g$  的 Hessian 矩阵. 忽略高阶无穷小量, 下一轮迭代的解  $\mathbf{x}^{(i+1)}$  应当为上述二次函数的极小值点. 于是对上式两边求梯度并代入  $\mathbf{x}^{(i+1)}$  可得

$$\mathbf{0} = \nabla g(\mathbf{x}^{(i+1)}) = \nabla g(\mathbf{x}^{(i)}) + \mathbf{H}_g(\mathbf{x}^{(i)}) (\mathbf{x}^{(i+1)} - \mathbf{x}^{(i)})$$

于是下一轮迭代的解即为线性方程组

$$\mathbf{H}_g(\mathbf{x}^{(i)}) (\mathbf{x}^{(i+1)} - \mathbf{x}^{(i)}) = -\nabla g(\mathbf{x}^{(i)})$$

的解. 对于弹簧质点系统, 认为  $g(\mathbf{x})$  的性质比较好, 只需一次迭代就能求得较为准确的解. 于是就有

$$\mathbf{H}_g(\mathbf{x}(t_k)) (\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)) = -\nabla g(\mathbf{x}(t_k))$$

根据  $g(\mathbf{x})$  的定义可知

$$\begin{aligned} \nabla g(\mathbf{x}(t_k)) &= \frac{1}{(\Delta t)^2} \mathbf{M}(\mathbf{x}(t_k) - \mathbf{y}(t_k)) + \nabla E(\mathbf{x}(t_k)) \\ \mathbf{H}_g(\mathbf{x}(t_k)) &= \frac{1}{(\Delta t)^2} \mathbf{M} + \mathbf{H}_E(\mathbf{x}(t_k)) \end{aligned}$$

其中  $\mathbf{H}_E(\mathbf{x})$  为能量函数  $E(\mathbf{x})$  的 Hessian 矩阵.

对于  $\nabla E(\mathbf{x}(t_k))$ , 我们可以把它写成对各个质点的位置  $\mathbf{x}_i$  的梯度  $\nabla_i E(\mathbf{x}) = \frac{\partial E(\mathbf{x})}{\partial \mathbf{x}_i}$  的堆叠向量, 并且已经知道质点  $i$  受到的内力之和等于能量的负梯度, 于是

$$\nabla E(\mathbf{x}(t_k)) = \begin{bmatrix} \nabla_1 E(\mathbf{x}) \\ \vdots \\ \nabla_n E(\mathbf{x}) \end{bmatrix}_{\mathbf{x}=\mathbf{x}(t_k)} = - \begin{bmatrix} \mathbf{f}_{1,\text{int}}(t_k) \\ \vdots \\ \mathbf{f}_{n,\text{int}}(t_k) \end{bmatrix}$$

接下来考虑  $\mathbf{H}_E(\mathbf{x}(t_k))$ . 根据弹簧质点模型的定义, 系统的总势能为各个弹簧势能之和:

$$E(\mathbf{x}) = \sum_{(i,j)} E_{ij}(\mathbf{x})$$

其中  $(i, j)$  表示以弹簧相连的质点. 于是

$$\mathbf{H}_E(\mathbf{x}) = \sum_{(i,j)} \mathbf{H}_{E_{ij}}(\mathbf{x})$$

其中  $\mathbf{H}_{E_{ij}}(\mathbf{x})$  为单个弹簧势能  $E_{ij}(\mathbf{x})$  的 Hessian 矩阵. 根据定义  $E_{ij} = \frac{1}{2}k_{ij}(|\mathbf{x}_j - \mathbf{x}_i| - l_{ij})^2$  有

$$\begin{aligned}\mathbf{H}_{ij} &:= \frac{\partial^2 E_{ij}(\mathbf{x})}{\partial \mathbf{x}_i^2} = k_{ij} \frac{(\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^t}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} + k_{ij} \left(1 - \frac{l_{ij}}{\|\mathbf{x}_i - \mathbf{x}_j\|}\right) \left(\mathbf{I} - \frac{(\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^t}{\|\mathbf{x}_i - \mathbf{x}_j\|^2}\right) \\ \frac{\partial^2 E_{ij}(\mathbf{x})}{\partial \mathbf{x}_j^2} &= \mathbf{H}_{ij}, \quad \frac{\partial^2 E_{ij}(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j} = -\mathbf{H}_{ij}\end{aligned}$$

对于  $E_{ij}(\mathbf{x})$  而言, 其值仅与  $\mathbf{x}$  中第  $i$  个向量  $\mathbf{x}_i$  和第  $j$  个向量  $\mathbf{x}_j$  有关. 于是将  $E_{ij}(\mathbf{x})$  的 Hessian 矩阵  $\mathbf{H}_{E_{ij}}(\mathbf{x})$  视作由  $N \times N$  个  $3 \times 3$  子矩阵组成的矩阵, 只有  $(i,i), (j,j), (i,j), (j,i)$  四个子矩阵非零, 且分别为  $\mathbf{H}_{ii}, \mathbf{H}_{jj}, -\mathbf{H}_{ij}, -\mathbf{H}_{ji}$ . 这就求得了  $\mathbf{H}_E$ , 于是再代入前述公式即可得隐式欧拉积分的迭代公式.

最后将隐式欧拉积分求解弹簧质点系统的过程总结如下:

**1.** 遍历系统中的所有弹簧, 然后进行如下两个操作:

a. 根据弹力的计算公式

$$\mathbf{f}_{ij} = -k_{ij}(|\mathbf{x}_i - \mathbf{x}_j| - l_{ij}) \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}$$

计算弹簧所连接的两个质点  $i, j$  的内力  $\mathbf{f}_{ij}, \mathbf{f}_{ji}$ , 并将其累加到质点  $i, j$  的总内力  $\mathbf{f}_{i,\text{int}}, \mathbf{f}_{j,\text{int}}$  上.

b. 根据弹簧的 Hessian 矩阵计算公式

$$\mathbf{H}_{ij} = \frac{\partial^2 E_{ij}(\mathbf{x})}{\partial \mathbf{x}_i^2} = k_{ij} \frac{(\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^t}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} + k_{ij} \left(1 - \frac{l_{ij}}{\|\mathbf{x}_i - \mathbf{x}_j\|}\right) \left(\mathbf{I} - \frac{(\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^t}{\|\mathbf{x}_i - \mathbf{x}_j\|^2}\right)$$

计算质点  $i, j$  对应的 Hessian 子矩阵, 并将其累加到系统总能量的 Hessian 矩阵  $\mathbf{H}_E$  的对应位置上.

完成上述两步后即可求出  $\nabla E$  和  $\mathbf{H}_E$ .

**2.** 根据  $\mathbf{y}(t_k)$  的定义

$$\mathbf{y}(t_k) = \mathbf{x}(t_k) + \Delta t \mathbf{v}(t_k) + (\Delta t)^2 \mathbf{M}^{-1} \mathbf{f}_{\text{ext}}$$

求出  $\mathbf{y}(t_k)$ .

**3.** 根据  $\nabla g(\mathbf{x}(t_k))$  和  $\mathbf{H}_g(\mathbf{x}(t_k))$  的定义

$$\nabla g(\mathbf{x}(t_k)) = \frac{1}{(\Delta t)^2} \mathbf{M}(\mathbf{x}(t_k) - \mathbf{y}(t_k)) + \nabla E(\mathbf{x}(t_k))$$

$$\mathbf{H}_g(\mathbf{x}(t_k)) = \frac{1}{(\Delta t)^2} \mathbf{M} + \mathbf{H}_E(\mathbf{x}(t_k))$$

求出  $\nabla g(\mathbf{x}(t_k))$  和  $\mathbf{H}_g(\mathbf{x}(t_k))$ .

**4.** 求解线性方程组

$$\mathbf{H}_g(\mathbf{x}(t_k))(\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)) = -\nabla g(\mathbf{x}(t_k))$$

得到  $\delta = \mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)$ .

## 5. 根据隐式欧拉的基本公式

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \boldsymbol{\delta}$$

$$\boldsymbol{v}(t_{k+1}) = \frac{\boldsymbol{\delta}}{\Delta t}$$

计算更新后的位置  $\boldsymbol{x}(t_{k+1})$  和速度  $\boldsymbol{v}(t_{k+1})$ . 如此就完成了一个时间间隔内的模拟.

因此, 依照上述过程的代码实现如下:

```

1 void AdvanceMassSpringSystem(MassSpringSystem & system, float const dt) {
2     // your code here: rewrite following code
3     size_t nPoints = system.Positions.size();
4     std::vector<Eigen::Triplet<float>> triplets_H_E;
5     std::vector<glm::vec3> vec_nabla_E(nPoints, glm::vec3(0.0f));
6     for (auto const spring : system.Springs) {
7         auto const i = spring.AdjIdx.first;
8         auto const j = spring.AdjIdx.second;
9         glm::vec3 const xi = system.Positions[i];
10        glm::vec3 const xj = system.Positions[j];
11        float d = glm::length(xi - xj);
12        glm::vec3 f = system.Stiffness * (glm::length(xi - xj) - spring.RestLength) * (xi -
13            xj) / d;
14        vec_nabla_E[i] += f;
15        vec_nabla_E[j] -= f;
16        glm::mat3 C = glm::outerProduct(xi - xj, xi - xj) / (d * d);
17        glm::mat3 I(1.0f);
18        glm::mat3 H = system.Stiffness * (C + (1.0f - spring.RestLength / d) * (I - C));
19        for (int col = 0; col < 3; ++col) {
20            for (int row = 0; row < 3; ++row) {
21                float val = H[col][row];
22                triplets_H_E.emplace_back(3 * i + row, 3 * i + col, val);
23                triplets_H_E.emplace_back(3 * j + row, 3 * j + col, val);
24                triplets_H_E.emplace_back(3 * i + row, 3 * j + col, -val);
25                triplets_H_E.emplace_back(3 * j + row, 3 * i + col, -val);
26            }
27        }
28        auto H_E = CreateEigenSparseMatrix(3 * nPoints, triplets_H_E);
29        auto nabla_E = glm2eigen(vec_nabla_E);
30        std::vector<glm::vec3> vec_x_k(nPoints, glm::vec3(0.0f));
31        std::vector<glm::vec3> vec_y_k(nPoints, glm::vec3(0.0f));
32        for (size_t i = 0; i < nPoints; ++i) {
33            vec_x_k[i] = system.Positions[i];

```

```

34     vec_y_k[i] = system.Positions[i] + dt * system.Velocities[i] + dt * dt * glm::vec3(0, -
35         system.Gravity, 0);
36
37     std::vector<Eigen::Triplet<float>> triplets_M;
38     for (size_t i = 0; i < 3 * nPoints; ++i) {
39         triplets_M.emplace_back(i, i, system.Mass);
40     }
41
42     auto M = CreateEigenSparseMatrix(3 * nPoints, triplets_M);
43
44
45     auto nabla_g = system.Mass * (x_k - y_k) / (dt * dt) + nabla_E;
46     auto H_g      = M / (dt * dt) + H_E;
47
48
49     std::vector<glm::vec3> vec_x_res = eigen2glm(ComputeSimplicialLLT(H_g, -nabla_g));
50     for (size_t i = 0; i < nPoints; ++i) {
51         if (system.Fixed[i]) continue;
52         system.Positions[i] += vec_x_res[i];
53         system.Velocities[i] = vec_x_res[i] / dt;
54     }
}

```

实现的效果图如下:

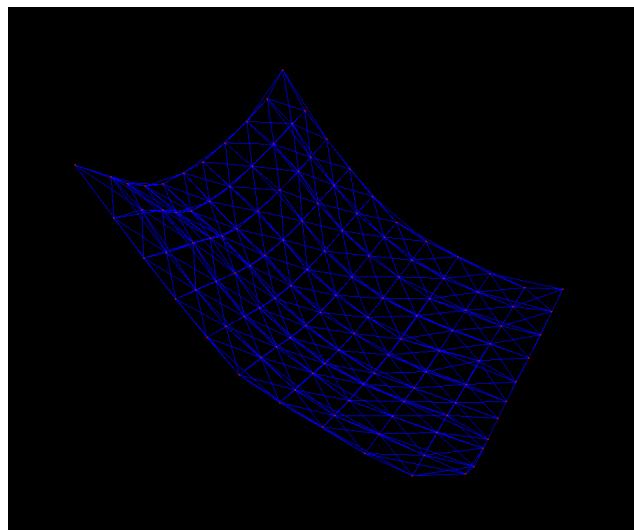


图 2: 使用隐式欧拉法求解弹簧质点模型的效果图