可视计算与交互概论-Lab4

xTryer

2024/11/23

1 Task 1: Inverse Kinematics

1.1 Sub-Task 1:forward kinematic

思路：从 startindex 开始更新 ik 上的关节位置信息，考虑 i 关节，其父关节为 i-1, 则有以下公式

$$global\_rotate[i] = global\_rotate[i-1] * local\_rotate[i]$$

$$global\_pos[i] = global\_pos[i-1] + global\_offset[i-1,i] = global\_rotate[i-1] * local\_offset[i-1,i]$$

即 i 关节的全局旋转等于 i-1 关节的全局旋转叠加上 i 关节相对 i-1 关节的局部旋转；i 关节的全局位置等于 i-1 关节的全局位置加上 i 关节相对 i-1 关节的偏移 (考虑旋转)。

```
void ForwardKinematics(IKSystem & ik, int StartIndex) {
    if (StartIndex == 0) {
        ik.JointGlobalRotation[0] = ik.JointLocalRotation[0];
        ik.JointGlobalPosition[0] = ik.JointLocalOffset[0];
        StartIndex                = 1;
    }

    for (int i = StartIndex; i < ik.JointLocalOffset.size(); i++) {
        // your code here: forward kinematics, update JointGlobalPosition and
    JointGlobalRotation
        ik.JointGlobalRotation[i] = glm::normalize(glm::normalize(ik.JointGlobalRotation[i -
    1])*glm::normalize(ik.JointLocalRotation[i]));
        glm::vec4 global_offset   = ik.JointGlobalRotation[i - 1] * glm::vec4(ik.
    JointLocalOffset[i], 1.0);
        ik.JointGlobalPosition[i] = ik.JointGlobalPosition[i - 1] + glm::vec3(global_offset);
    }
}
```

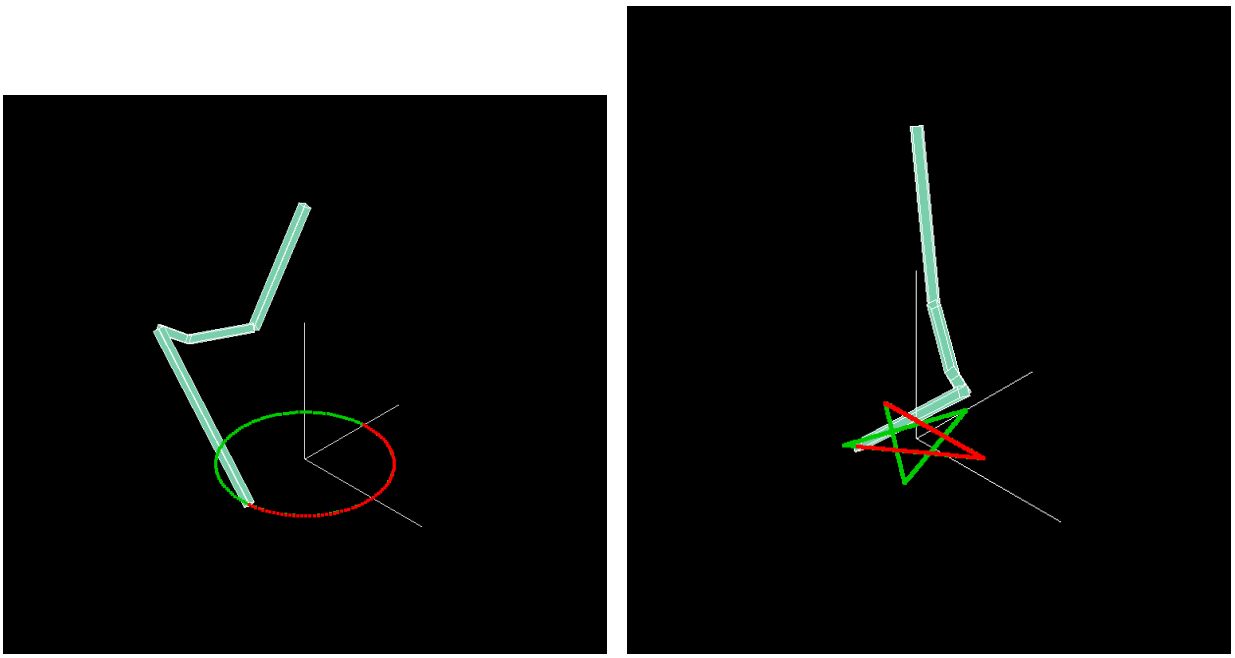
1.2 Sub-Task 2:CCD IK

思路：对于 CCD IK，想法是从尾部节点开始，每次旋转都使得自己、末端关节节点、目标位置三者位于同一条直线上，这样使得每一次旋转后末端关节节点都处在与目标位置最近的位置 (对于目前正在处理的关节节点的所有可能局部旋转而言)。

具体实现：对每一个关节节点，分别求其到 EndPosition 与 EndEffectorPosition 的向量，然后标准化得到 cur_to_tar 与 cur_to_effect, 然后使用 glm::rotation 函数计算从 cur_to_tar 旋转至 cur_to_effect 的旋转四元数，并更新相应节点的局部旋转值，最后从这个节点开始对其子节点进行位姿更新，通过不断的迭代实现末端关节节点逼近目标位置。

```
void InverseKinematicsCCD(IKSystem & ik, const glm::vec3 & EndPosition, int maxCCDIKIteration,
    float eps) {
    ForwardKinematics(ik, 0);
    // These functions will be useful: glm::normalize, glm::rotation, glm::quat * glm::quat
    for (int CCDIKIteration = 0; CCDIKIteration < maxCCDIKIteration && glm::l2Norm(ik.
   EndEffectorPosition() - EndPosition) > eps; CCDIKIteration++) {
        // your code here: ccd ik
        for (int i = ik.NumJoints() - 1; i >= 0; i--)
        {
            glm::vec3 cur_joint_pos = ik.JointGlobalPosition[i];
            glm::vec3 end_effector_pos = ik.EndEffectorPosition();

            glm::vec3 cur_to_tar = glm::normalize(EndPosition - cur_joint_pos);
            glm::vec3 cur_to_effect = glm::normalize(end_effector_pos - cur_joint_pos);


            glm::quat rotation_quat = glm::rotation(cur_to_effect,cur_to_tar);
            ik.JointLocalRotation[i] = rotation_quat * ik.JointLocalRotation[i];

            ForwardKinematics(ik, i);


        }
    }
}
```

得到效果图如下：



(a) CCD-IK:circle (b) CCD-IK:star

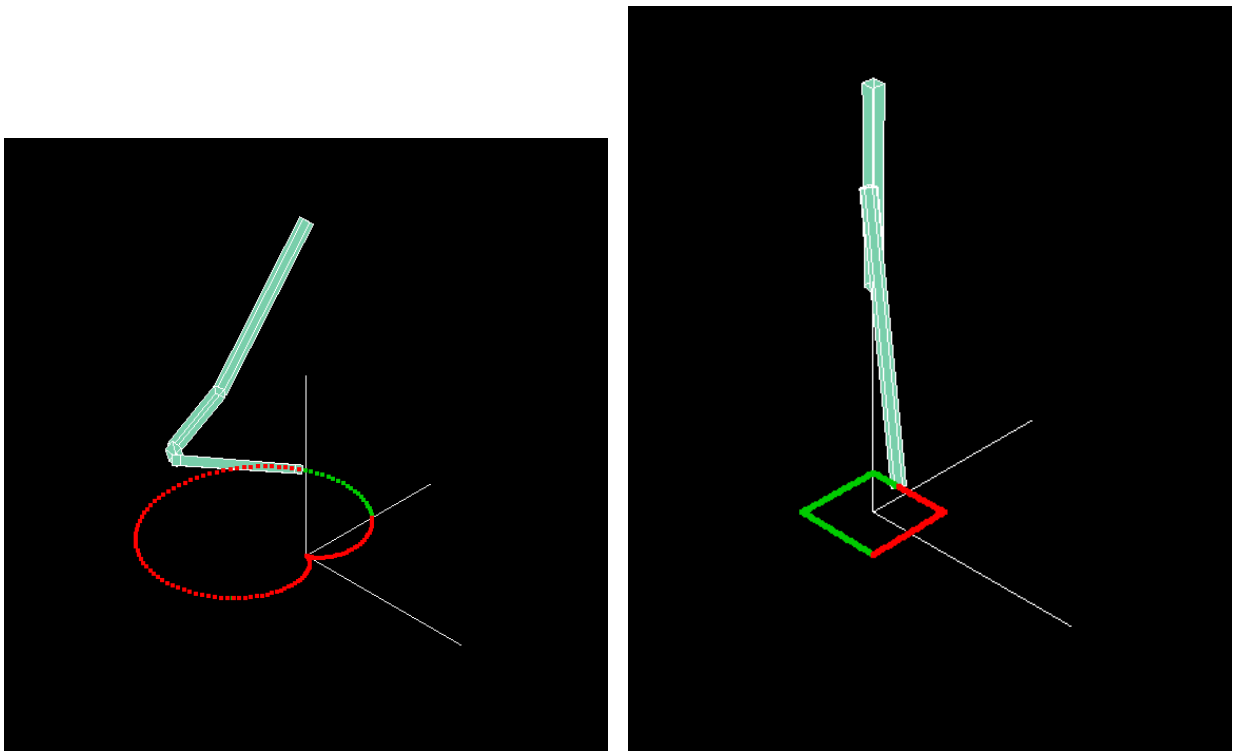
图 1: Sub-Task 2:CCD IK-results

1.3 Sub-Task 3:FABR IK

思路：与 CCD IK 不同，FABR IK 不通过计算旋转角度然后调整位姿，而是直接调整骨骼方向，改变关节位置。整个过程经历了两次更新，在 backward update 阶段，先假设能够末端关节位置位于目标位置，然后在此基础上根据 cur_joint 的局部 offset 和 next_position 到 cur_joint 的方向更新 cur_joint 的位置，直至更新得到一个新的根节点位置 p0'；然后进行 forward update, 保持 p0 位置不变，利用 p0',p1' 等 backward update 得到的新关节位置计算得到新的关节位置 p0",p1"..... 经过这样两次 update，可以使得末端关节位置不断逼近目标位置。

```
void InverseKinematicsFABR(IKSystem & ik, const glm::vec3 & EndPosition, int maxFABRIKIteration, float eps) {
    ForwardKinematics(ik, 0);
    int nJoints = ik.NumJoints();
    std::vector<glm::vec3> backward_positions(nJoints, glm::vec3(0, 0, 0)), forward_positions(nJoints, glm::vec3(0, 0, 0));
    for (int IKIteration = 0; IKIteration < maxFABRIKIteration && glm::l2Norm(ik.EndEffectorPosition() - EndPosition) > eps; IKIteration++) {
        // task: fabr ik
        // backward update
        glm::vec3 next_position = EndPosition;
        backward_positions[nJoints - 1] = EndPosition;

        for (int i = nJoints - 2; i >= 0; i--) {
            // your code here
            glm::vec3 normalized_dir = glm::normalize(ik.JointGlobalPosition[i] - next_position);
            backward_positions[i] = next_position + normalized_dir * glm::length(ik.JointLocalOffset[i + 1]);
            next_position = backward_positions[i];
        }
```

```
17
18          // forward update
19          glm::vec3 now_position = ik.JointGlobalPosition[0];
20          forward_positions[0] = ik.JointGlobalPosition[0];
21          for (int i = 0; i < nJoints - 1; i++) {
22              // your code here
23              glm::vec3 normalized_dir = glm::normalize(backward_positions[i + 1] - now_position
    );
24              forward_positions[i + 1] = now_position + normalized_dir * glm::length(ik.
    JointLocalOffset[i + 1]);
25              now_position             = forward_positions[i + 1];
26          }
27          ik.JointGlobalPosition = forward_positions; // copy forward positions to
    joint_positions
28      }
29
30      // Compute joint rotation by position here.
31      for (int i = 0; i < nJoints - 1; i++) {
32          ik.JointGlobalRotation[i] = glm::rotation(glm::normalize(ik.JointLocalOffset[i + 1]),
    glm::normalize(ik.JointGlobalPosition[i + 1] - ik.JointGlobalPosition[i]));
33      }
34      ik.JointLocalRotation[0] = ik.JointGlobalRotation[0];
35      for (int i = 1; i < nJoints - 1; i++) {
36          ik.JointLocalRotation[i] = glm::inverse(ik.JointGlobalRotation[i - 1]) * ik.
    JointGlobalRotation[i];
37      }
38      ForwardKinematics(ik, 0);
39  }
```



(a) CCD-FABR:heart (b) CCD-IK:square

图 2: Sub-Task 3:FABR IK-results

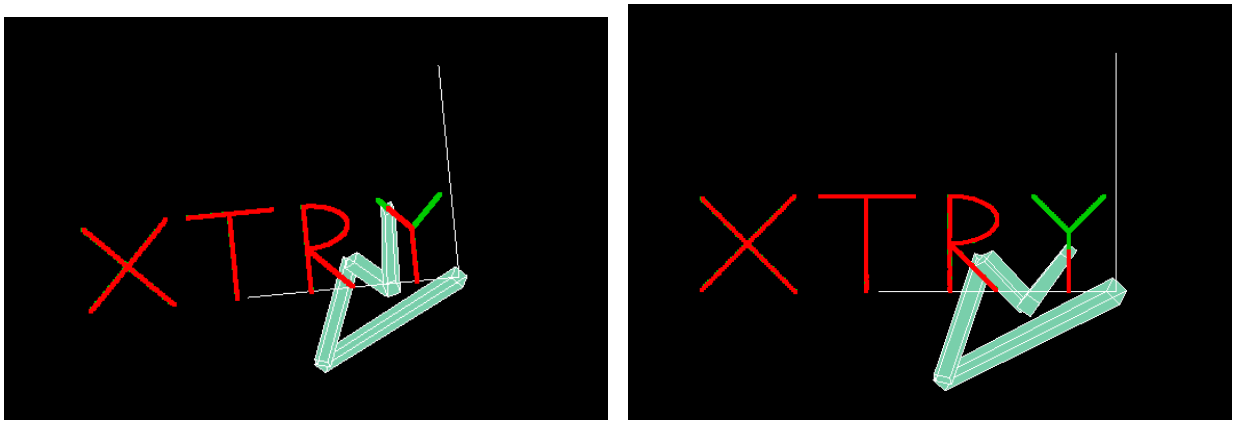
1.4 Sub-Task 4:Custom Draw

思路：类似参数方程，画直线使用一次函数，画曲线可以使用圆/椭圆的参数方程。最终得到 custom str:XTRY。

```
IKSystem::Vec3ArrPtr IKSystem::BuildCustomTargetPosition() {
    int nums      = 50;
    using Vec3Arr = std::vector<glm::vec3>;
    std::shared_ptr<Vec3Arr> custom(new Vec3Arr(nums*12));
    int                      index = 0;

    // X
    for (int i = 0; i < nums; i++) {
        float x_val        = 0.4 * i / nums;
        float y_val        = 0.4*i/nums;
        (*custom)[index++] = glm::vec3(x_val+1.35, 0.0f, y_val);
    }
    for (int i = 0; i < nums; i++) {
        float x_val        = 0.4 * i / nums;
        float y_val        = 0.4-0.4 * i / nums;
        (*custom)[index++] = glm::vec3(x_val+1.35, 0.0f, y_val);
    }

    // T
    for (int i = 0; i < nums; i++) {
        float x_val        = 0.4 * i / nums;
        float y_val        = 0.4;
        (*custom)[index++] = glm::vec3(x_val+0.85, 0.0f, y_val);
    }
```

```
    for (int i = 0; i < nums; i++) {
        float x_val         = 0.2;
        float y_val         = 0.4 * i / nums;
        (*custom)[index++] = glm::vec3(x_val+0.85, 0.0f, y_val);
    }
        //R
        for (int i = 0; i < nums; i++) {
        float x_val         = 0.2;
        float y_val         = 0.4 * i / nums;
        (*custom)[index++] = glm::vec3(x_val+0.5, 0.0f, y_val);
    }
        for (int i = 0; i < nums; i++) {
        float x_val         = 0.2 - 0.2 * sin(EIGEN_PI * i / nums);
            float y_val         = 0.3+0.1 * cos(EIGEN_PI* i / nums);
            (*custom)[index++] = glm::vec3(x_val+0.5, 0.0f, y_val);
        }
        for (int i = 0; i < nums; i++) {
            float x_val         = 0.2-0.2*i/nums;
            float y_val         = 0.2-0.2*i/nums;
            (*custom)[index++] = glm::vec3(x_val+0.5, 0.0f, y_val);
        }
    // Y
        for (int i = 0; i < nums; i++) {
            float x_val         = 0.15;
            float y_val         = 0.25*i/nums;
            (*custom)[index++] = glm::vec3(x_val+0.05, 0.0f, y_val);
        }
        for (int i = 0; i < nums; i++) {
            float x_val         = 0.15+0.15*i/nums;
            float y_val         = 0.25+0.15 * i / nums;
            (*custom)[index++] = glm::vec3(x_val+0.05, 0.0f, y_val);
        }
        for (int i = 0; i < nums; i++) {
            float x_val         = 0.15 - 0.15 * i / nums;
            float y_val         = 0.25 + 0.15 * i / nums;
            (*custom)[index++] = glm::vec3(x_val+0.05, 0.0f, y_val);
        }
    custom->resize(index);
    return custom;
}
```

得到效果图如下:



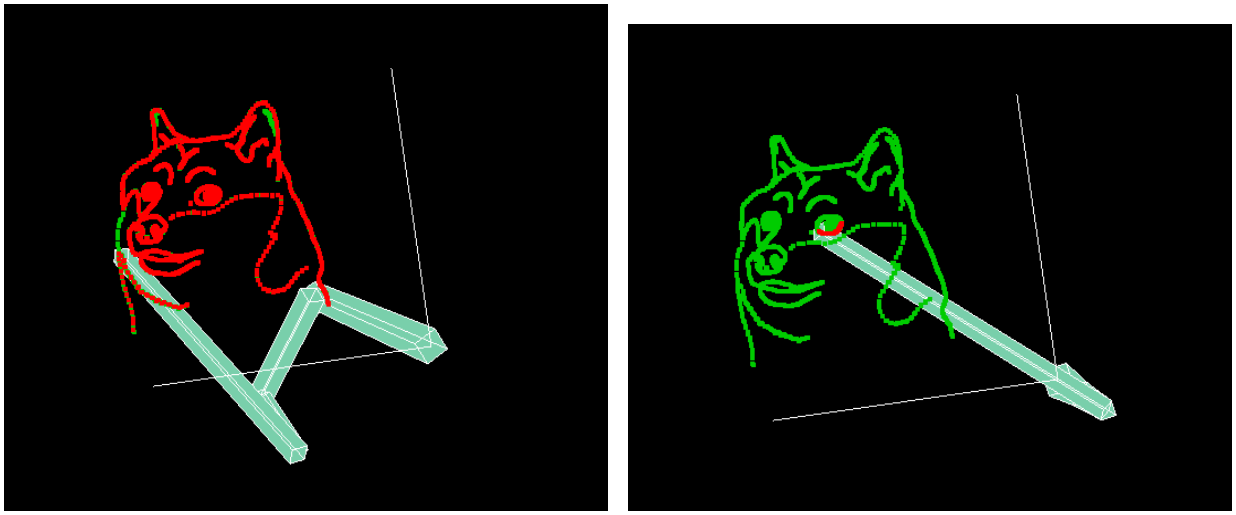
(a) Cutom Draw-XTRY (b) Cutom Draw-XTRY

图 3: Sub-Task 4:Cutom Draw-XTRY

彩蛋-真 • doge，在 wolframalpha 网站上发现了 doge-curve, 修改了其中的参数格式，得到 doge :D

```
    float doge_custom_x(float t) {
        float x_t = ***//too long->hidden
        return x_t+550.0;
    }
    float doge_custom_y(float t) {
        float y_t = ***//too long->hidden
        return y_t+550.0;
    }
    IKSystem::Vec3ArrPtr IKSystem::BuildCustomTargetPosition() {

    int nums       = 5000;
    using Vec3Arr = std::vector<glm::vec3>;
    std::shared_ptr<Vec3Arr> custom(new Vec3Arr(nums));
    int                      index = 0;
    for (int i = 0; i < nums; i++) {
        float x_val = 1.5e-3f * doge_custom_x(92 * glm::pi<float>() * i / nums);
        float y_val = 1.5e-3f * doge_custom_y(92 * glm::pi<float>() * i / nums);
        if (std::abs(x_val) < 1e-3 || std::abs(y_val) < 1e-3) continue;
        (*custom)[index++] = glm::vec3(1.6f - x_val, 0.0f, y_val - 0.2f);
    }
    custom->resize(index);
    return custom;

}
```

得到效果图如下:

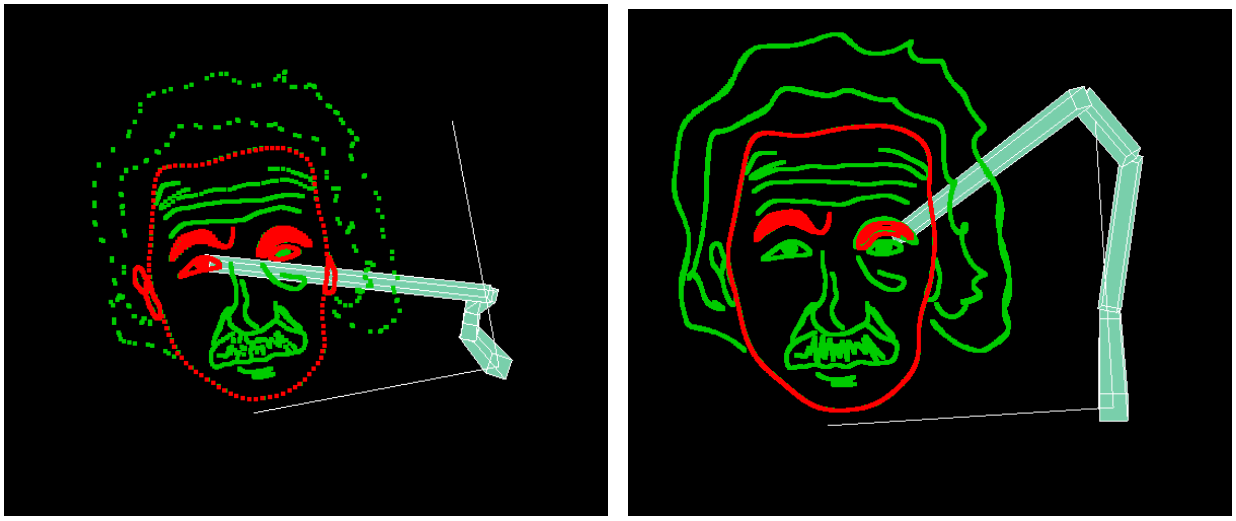


(a) Cutom Draw-DOGE (b) Cutom Draw-DOGE

图 4: Sub-Task 4:Cutom Draw-DOGE

1.5 Sub-Task 4-1:Cutom Draw sample

思路：采样点不均匀的原因可能是在头发处坐标变化比较剧烈，如果采样密度不够高，就会导致头发区域的采样点比较稀疏，一个可行的方法是增加采样密度, 下图分别是总采样 num=5000 与总采样 num=50000 的效果图区别：



(a) sample num=5000 (b) sample num=50000

图 5: Sub-Task 4-1:Cutom Draw-sample

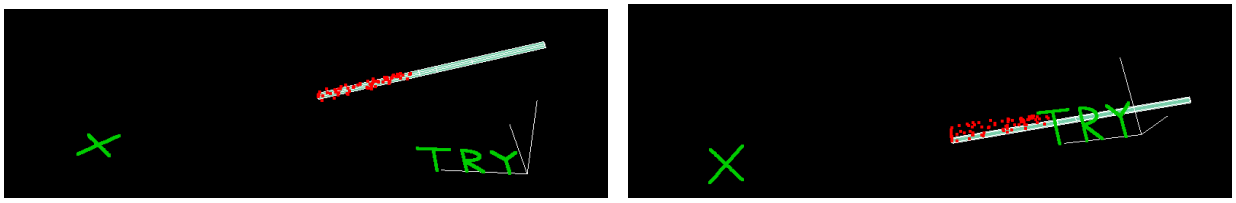
但这只是增加了总体的采样数目，会导致计算开销较大，一个可行的方案是根据图像的剧烈变化程度动态调整采样数目，在图像剧烈变化的地方 (可以用梯度图分析变化程度) 提高采样密度, 这样就可以使得采样点更加均匀。

1.6 Report

1.6.1 1. 如果目标位置太远，无法到达，IK 结果会怎样？

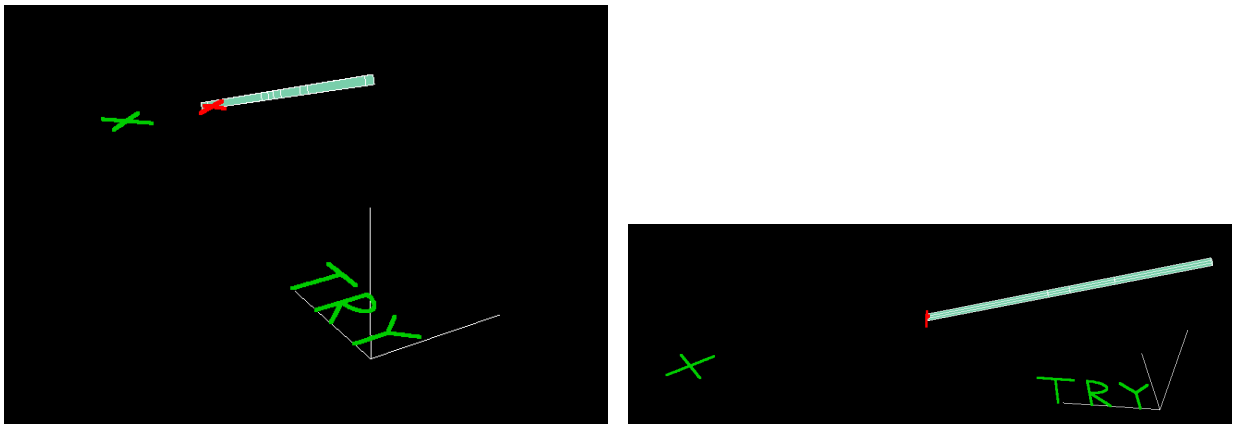
会把“臂”伸直指向目标位置，然后在 origin_pos->tar_pos 方向上的最远处作画. 其中 CCD-IK 会抽搐，留下无规律的散点，而 FABR-IK 会在最远处画出对应图像

如下图所示 (把 XTRY 中的 X 拉远至臂长无法到达的位置):



(a) CCD-IK:Too Far (b) CCD-IK:Too Far

图 6: Task 1-Question 1-CCD-IK



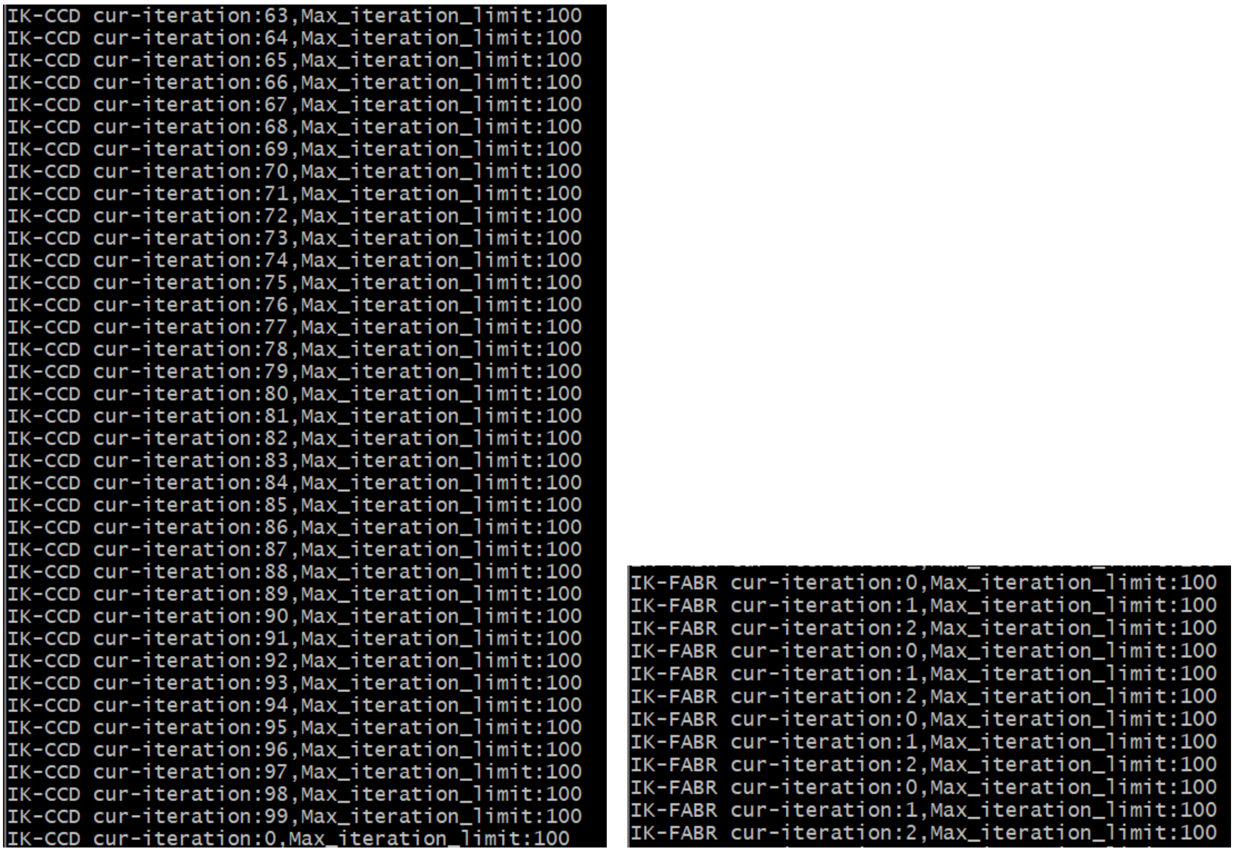
(a) FABR-IK:Too Far (b) FABR-IK:Too Far

图 7: Task 1-Question 1-FABR-IK

1.6.2 2. 比较 CCD IK 和 FABR IK 所需要的迭代次数。

FABR IK 的迭代次数比 CCD IK 的迭代次数更少。

如下图是画 circle 时 CCD-IK 与 FABR-IK 的迭代输出结果,可以看出 CCD-IK 基本上到 max_iteration=100 都还没有收敛，而 FABR-IK 只需要迭代个位数次就能早早完成收敛。



(a) CCD-IK　　(b) FABR-IK

图 8: Task 1-Question 2-iterations time results

1.6.3 3. IK 多解导致前后两帧关节旋转抖动的情况

可能的解决方案：限制旋转角度，规定 IK 旋转只能在一个规定的小范围内进行。

2 Task 2: Mass-Spring System

整体思路：
对于 Implicit Euler:

$$x_{n+1} = x_n + dt \cdot v_{n+1}$$
$$v_{n+1} = v_n + dt \cdot M^{-1} f(x_{n+1})$$

经过分析计算，我们得到 $x_{n+1} = (x_n + dt(v_n + dt \cdot M^{-1} f_{ext})) + dt^2 M^{-1} f_{int}(x_{n+1})$
求解这个方程相当于求解 $x_{n+1} = argmin_x g(x), g(x) = \frac{1}{2dt^2}|x-y|_M^2 + E(x)$
其中 $|x-y|_M^2 = (x-y)^T M(x-y), y = x_n + dt(v_n + dt \cdot M^{-1} f_{ext})$
进一步，我们对 g(x) 在当前尝试解 x_i 处左二阶泰勒展开，得到

$$g(x) = g(x_i) + \bigtriangledown g(x_i) \cdot (x - x_i) + \frac{1}{2}(x - x_i)^T H_g(x_i)(x - x_i) + O(||x - x - i||^3)$$

其中，$\bigtriangledown g(x_i)$ 表示 $g(x_i)$ 的梯度，$H_g(x_i)$ 表示 $g(x)$ 在 x_i 处的海瑟矩阵。
然后我们忽略三阶小量，对式子两边求梯度，再代入 x_{i+1}, 得到

$$\bigtriangledown g(x_{i+1}) = \bigtriangledown g(x_i) + H_g(x_i)(x_{i+1} - x_i)$$

令这条等于 0(求极值点), 得到 $x_{i+1} = x_i - H_g^{-1}(x_i) \bigtriangledown g(x_i)$

现在我们的目的是求出 $\bigtriangledown g(x_i)$ 与 $H_g(x_i)$, 我们有

$$\bigtriangledown g(x_i) = \frac{1}{dt^2}M(x_i - y_i) + \bigtriangledown E(x_i)$$
$$H_g(x_i) = \frac{1}{dt^2}M + H(x_k)$$

其中 $H(x_k)$ 是弹性势能 E(x) 的海瑟矩阵，有 $H(x_k) = \Sigma_{(i,j)}H_{ij}(x_k)$
在完成 $\bigtriangledown g(x_i)$ 与 $H_g(x_i)$ 的求解后，我们得到 $x_{i+1} - x_i = -H_g^{-1}(x_i) \bigtriangledown g(x_i)$
然后更新 x_{i+1} 并以此为基础更新 v_{i+1}
具体实现：
首先初始化 x0,v0,f0(外力矩阵, 此处为简单的重力加速度 mg/m), 然后根据公式计算 y0

```
int pos_num = system.Positions.size();
// initialize
Eigen::VectorXf x0     = glm2eigen(system.Positions);
Eigen::VectorXf v0 = glm2eigen(system.Velocities);
Eigen::VectorXf f0_extern_grav = glm2eigen(std::vector<glm::vec3>(system.Positions.size(),
                                    glm::vec3(0, -system.Gravity, 0)));
Eigen::VectorXf system_m_vectorxf = glm2eigen(std::vector<glm::vec3>(system.Positions.size(),
                  glm::vec3(system.Mass,system.Mass,system.Mass)));

Eigen::VectorXf y0 = x0 + ddt * (v0 + ddt * f0_extern_grav);
```

利用函数 eval_grad_g 计算出 grad_g, 主要围绕公式 $\bigtriangledown g(x_i) = \frac{1}{dt^2}M(x_i - y_i) + \bigtriangledown E(x_i)$, 其中 $\bigtriangledown E(x_i)$ 总能量关于所有质点的位置的梯度就是把关于每个质点的梯度拼接起来，而能量关于质点的梯度相当于质点受到的力 (取负号表示方向相反)。

```
Eigen::VectorXf eval_intern_f(MassSpringSystem & system, Eigen::VectorXf & x_vec)
{
    std::vector<glm::vec3> forces(system.Positions.size(), glm::vec3(0));
    std::vector<glm::vec3> x_vec3 = eigen2glm(x_vec);
    for (auto const spring : system.Springs) {
        auto const       p0  = spring.AdjIdx.first;
        auto const       p1  = spring.AdjIdx.second;
        if (system.Fixed[p0] && system.Fixed[p1]) continue;
        glm::vec3 const x01 = x_vec3[p1] - x_vec3[p0]; // use updated x_vec

        glm::vec3 const e01 = glm::normalize(x01);
        glm::vec3       f   = (system.Stiffness * (glm::length(x01) - spring.RestLength)) *
    e01;
        forces[p0] += f;
        forces[p1] -= f;
    }
    return glm2eigen(forces);
}
Eigen::VectorXf eval_grad_g(MassSpringSystem & system, Eigen::VectorXf & x_vec, Eigen::
    VectorXf & y_vec, float ddt)
{
    Eigen::VectorXf grad_g_ret = system.Mass * (x_vec - y_vec) / (ddt*ddt);
```

```
    grad_g_ret -= eval_intern_f(system,x_vec);
    return grad_g_ret;
}
```

进一步，我们计算 $H_g(x_i)$, 主要围绕公式 $H_g(x_i) = \frac{1}{dt^2}M + H(x_k)$

前面带 M 的部分好处理，直接给 SparseMatrix 对角线赋值即可

```
// M Matrix
std::vector<Eigen::Triplet<float>> m_triplets;
float                              tmp_m = system.Mass / (ddt * ddt);
for (int i = 0; i < 3*pos_num; i++)
{
    m_triplets.emplace_back(i, i,tmp_m);
}
Eigen::SparseMatrix<float> m_mat = CreateEigenSparseMatrix(size_t(3) * pos_num,m_triplets);
```

后面的 Hessian 矩阵部分利用公式

$$H_{e-ij} = k_{ij}\frac{(x_i - x_j)(x_i - x_j)^T}{||x_i - x_j||^2} + k_{ij}(1 - \frac{l_{ij}}{||x_i - x_j||})(I - \frac{(x_i - x_j)(x_i - x_j^T)}{||x_i - x_j||^2})$$

得到一个 3*3 的 H_{e-ij} 小矩阵，然后进一步计算对应的 $H_{ij}(x_k)$ 大矩阵，其 shape 为 $(n*n)*(3*3)$, 即 n*n 个 3*3 的块，其中 $H_{ij}(i,j) = H_{ij}(j,i) = -H_{e-ij}$,$H_{ij}(i,i) = H_{ij}(j,j) = H_{e-ij}$

```
// Hessian Matrix
std::vector<Eigen::Triplet<float>> Hessian_triplets;
std::vector<glm::vec3>             x_vec3 = eigen2glm(x_vec);
for (auto const spring : system.Springs) {
    auto const p0 = spring.AdjIdx.first;
    auto const p1 = spring.AdjIdx.second;
    glm::vec3 const x01 = x_vec3[p1] - x_vec3[p0];

    float len_x01 = glm::length(x01);

    glm::mat3 identity_matrix {1.0f};

    glm::mat3 x_matrix(0.0f);

    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            x_matrix[i][j] = x01[i] * x01[j];
        }
    }

    glm::mat3 hessian_tmp_matrix = x_matrix / (len_x01*len_x01);
    hessian_tmp_matrix += (1.0f - spring.RestLength / len_x01) * (identity_matrix - x_matrix/(
    len_x01*len_x01));
    hessian_tmp_matrix *= system.Stiffness;
```

```
26
27      for (int i = 0; i < 3; i++)
28      {
29          for (int j = 0; j < 3; j++)
30          {
31              Hessian_triplets.emplace_back(p0 * 3 + i, p0 * 3 + j, hessian_tmp_matrix[i][j]);
    //Warning: from size_t to
32              Hessian_triplets.emplace_back(p1 * 3 + i, p1 * 3 + j, hessian_tmp_matrix[i][j]);
33              Hessian_triplets.emplace_back(p0 * 3 + i, p1 * 3 + j, -hessian_tmp_matrix[i][j]);
34              Hessian_triplets.emplace_back(p1 * 3 + i, p0 * 3 + j, -hessian_tmp_matrix[i][j]);
35          }
36      }
37
38  }
39
40  Eigen::SparseMatrix<float> Hessian_mat = CreateEigenSparseMatrix(3 * pos_num, Hessian_triplets
    );
```

最后把得到的 m_mat 与 Hessian_mat 相加即得到待求的 $H_g(x_i)$

完整的 eval_grad2_g 函数如下：

```
1   void eval_grad2_g(MassSpringSystem & system, Eigen::VectorXf & x_vec, float ddt, Eigen::
    SparseMatrix<float> & dst_matrix)
2   {
3       int pos_num = system.Positions.size();
4
5       Eigen::SparseMatrix<float> grad2_g_ret(3 * pos_num, 3 * pos_num);
6
7       // M Matrix
8       std::vector<Eigen::Triplet<float>> m_triplets;
9       float                              tmp_m = system.Mass / (ddt * ddt);
10      for (int i = 0; i < 3*pos_num; i++)
11      {
12          m_triplets.emplace_back(i, i,tmp_m);
13      }
14      Eigen::SparseMatrix<float> m_mat = CreateEigenSparseMatrix(size_t(3) * pos_num,m_triplets)
    ;
15
16      // Hessian Matrix
17      std::vector<Eigen::Triplet<float>> Hessian_triplets;
18      std::vector<glm::vec3>             x_vec3 = eigen2glm(x_vec);
19      for (auto const spring : system.Springs) {
20          auto const p0 = spring.AdjIdx.first;
21          auto const p1 = spring.AdjIdx.second;
22          if (system.Fixed[p0] && system.Fixed[p1]) continue;
23          glm::vec3 const x01 = x_vec3[p1] - x_vec3[p0];
24
25          float len_x01 = glm::length(x01);
26
27          glm::mat3 identity_matrix {1.0f};
```

```
        //printf("%f %f %f\n", identity_matrix[0][0], identity_matrix[1][1], identity_matrix[2][2]);

        glm::mat3 x_matrix(0.0f);

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                x_matrix[i][j] = x01[i] * x01[j];
            }
        }

        glm::mat3 hessian_tmp_matrix = x_matrix / (len_x01*len_x01);
        hessian_tmp_matrix += (1.0f - spring.RestLength / len_x01) * (identity_matrix - x_matrix/(len_x01*len_x01));
        hessian_tmp_matrix *= system.Stiffness;

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                Hessian_triplets.emplace_back(p0 * 3 + i, p0 * 3 + j, hessian_tmp_matrix[i][j]); //Warning: from size_t to
                Hessian_triplets.emplace_back(p1 * 3 + i, p1 * 3 + j, hessian_tmp_matrix[i][j]);
                Hessian_triplets.emplace_back(p0 * 3 + i, p1 * 3 + j, -hessian_tmp_matrix[i][j]);
                Hessian_triplets.emplace_back(p1 * 3 + i, p0 * 3 + j, -hessian_tmp_matrix[i][j]);
            }
        }

    }

    Eigen::SparseMatrix<float> Hessian_mat = CreateEigenSparseMatrix(3 * pos_num, Hessian_triplets);

    dst_matrix = m_mat+Hessian_mat;
}
```

然后我们利用得到的 grad2_g 与 grad_g 列出方程 $H_g(x_i)(x_{i+1} - x_i) = - \bigtriangledown g(x_i)$ 调用 ComputeSimplicialLLT 计算得 δx, 然后更新 x_{i+1} 和 v_{i+1} 即可。

```
Eigen::VectorXf  grad_g = eval_grad_g(system, x0, y0, ddt);
Eigen::SparseMatrix<float> grad2_g(3 * pos_num, 3 * pos_num);
eval_grad2_g(system, x0, ddt, grad2_g);

Eigen::VectorXf x1_x0 = ComputeSimplicialLLT(grad2_g, -grad_g);

```

```
Eigen::VectorXf  final_x   = x1_x0 + x0;
std::vector<glm::vec3> final_x_vec3 = eigen2glm(final_x);
std::vector<glm::vec3> final_v = eigen2glm(v0 + ddt * (eval_intern_f(system,final_x)/system.
    Mass + f0_extern_grav));

for (std::size_t i = 0; i < system.Positions.size(); i++) {
    if (system.Fixed[i]) continue;
    system.Velocities[i] = final_v[i];
    system.Positions[i]  = final_x_vec3[i];
}
```

完整代码：

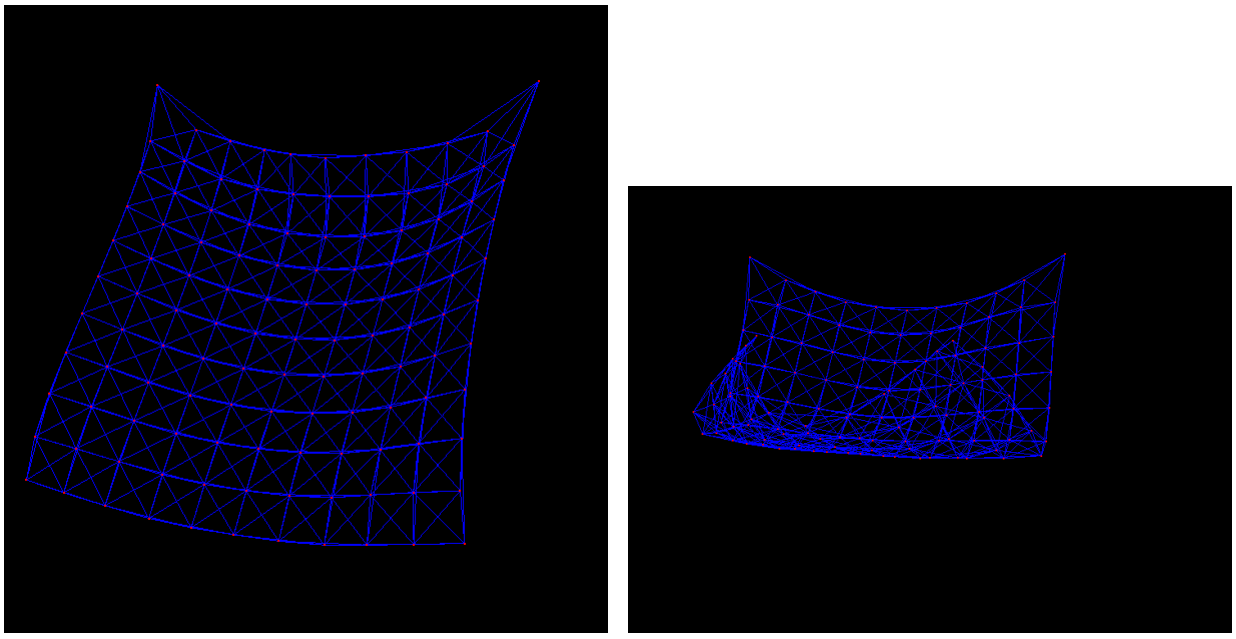
```
float eval_g(MassSpringSystem & system, Eigen::VectorXf & x_vec, Eigen::VectorXf & y_vec,float
    ddt)
{
    Eigen::VectorXf tmp_vec = x_vec-y_vec;
    float           eval_g_ret = ((tmp_vec * system.Mass).dot(tmp_vec))/ (2.0f*ddt * ddt);
    for (auto const spring : system.Springs)
    {

        auto const p0 = spring.AdjIdx.first;
        auto const p1 = spring.AdjIdx.second;
        if (system.Fixed[p0] && system.Fixed[p1]) continue;

        glm::vec3 const x01 = system.Positions[p1] - system.Positions[p0];
        float           deta_length = glm::length(x01) - spring.RestLength;
        float           tmp_E = 1.0f / 2.0f * system.Stiffness * deta_length*deta_length;
        eval_g_ret += tmp_E;
    }
    return eval_g_ret;
}

Eigen::VectorXf eval_intern_f(MassSpringSystem & system, Eigen::VectorXf & x_vec)
{
    std::vector<glm::vec3> forces(system.Positions.size(), glm::vec3(0));
    std::vector<glm::vec3> x_vec3 = eigen2glm(x_vec);
    for (auto const spring : system.Springs) {
        auto const      p0  = spring.AdjIdx.first;
        auto const      p1  = spring.AdjIdx.second;
        if (system.Fixed[p0] && system.Fixed[p1]) continue;
        glm::vec3 const x01 = x_vec3[p1] - x_vec3[p0]; // use updated x_vec

        glm::vec3 const e01 = glm::normalize(x01);
        glm::vec3       f   = (system.Stiffness * (glm::length(x01) - spring.RestLength)) *
    e01;
        forces[p0] += f;
        forces[p1] -= f;
    }
    return glm2eigen(forces);
```

```
}
Eigen::VectorXf eval_grad_g(MassSpringSystem & system, Eigen::VectorXf & x_vec, Eigen::
    VectorXf & y_vec, float ddt)
{
    Eigen::VectorXf grad_g_ret = system.Mass * (x_vec - y_vec) / (ddt*ddt);
    grad_g_ret -= eval_intern_f(system,x_vec);
    return grad_g_ret;
}

void eval_grad2_g(MassSpringSystem & system, Eigen::VectorXf & x_vec, float ddt, Eigen::
    SparseMatrix<float> & dst_matrix)
{
    int pos_num = system.Positions.size();

    Eigen::SparseMatrix<float> grad2_g_ret(3 * pos_num, 3 * pos_num);

    // M Matrix
    std::vector<Eigen::Triplet<float>> m_triplets;
    float                              tmp_m = system.Mass / (ddt * ddt);
    for (int i = 0; i < 3*pos_num; i++)
    {
        m_triplets.emplace_back(i, i,tmp_m);
    }
    Eigen::SparseMatrix<float> m_mat = CreateEigenSparseMatrix(size_t(3) * pos_num,m_triplets)
    ;

    // Hessian Matrix
    std::vector<Eigen::Triplet<float>> Hessian_triplets;
    std::vector<glm::vec3>             x_vec3 = eigen2glm(x_vec);
    for (auto const spring : system.Springs) {
        auto const p0 = spring.AdjIdx.first;
        auto const p1 = spring.AdjIdx.second;
        if (system.Fixed[p0] && system.Fixed[p1]) continue;
        glm::vec3 const x01 = x_vec3[p1] - x_vec3[p0];

        float len_x01 = glm::length(x01);

        glm::mat3 identity_matrix {1.0f};
        //printf("%f %f %f\n", identity_matrix[0][0], identity_matrix[1][1], identity_matrix
    [2][2]);

        glm::mat3 x_matrix(0.0f);

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                x_matrix[i][j] = x01[i] * x01[j];
            }
        }

```

```
83          glm::mat3 hessian_tmp_matrix = x_matrix / (len_x01*len_x01);
84          hessian_tmp_matrix += (1.0f - spring.RestLength / len_x01) * (identity_matrix -
    x_matrix/(len_x01*len_x01));
85          hessian_tmp_matrix *= system.Stiffness;
86
87          for (int i = 0; i < 3; i++)
88          {
89              for (int j = 0; j < 3; j++)
90              {
91                  Hessian_triplets.emplace_back(p0 * 3 + i, p0 * 3 + j, hessian_tmp_matrix[i][j
    ]); //Warning: from size_t to
92                  Hessian_triplets.emplace_back(p1 * 3 + i, p1 * 3 + j, hessian_tmp_matrix[i][j
    ]);
93                  Hessian_triplets.emplace_back(p0 * 3 + i, p1 * 3 + j, -hessian_tmp_matrix[i][j
    ]);
94                  Hessian_triplets.emplace_back(p1 * 3 + i, p0 * 3 + j, -hessian_tmp_matrix[i][j
    ]);
95              }
96          }
97
98      }
99
100     Eigen::SparseMatrix<float> Hessian_mat = CreateEigenSparseMatrix(3 * pos_num,
    Hessian_triplets);
101
102     dst_matrix = m_mat+Hessian_mat;
103 }
104
105 void AdvanceMassSpringSystem(MassSpringSystem & system, float const dt) {
106     // Implicit Euler algorithm
107     // x_{k+1}=x_{k}+ddt*v_{k+1}
108     // v_{k+1}=v_{k}+ddt*F(x_{k+1})/M
109     int const   steps = 1;
110     float const ddt   = dt / steps;
111     // without extern iterations
112     for (std::size_t s = 0; s < steps; s++) {
113         int pos_num = system.Positions.size();
114         // initialize
115         Eigen::VectorXf x0    = glm2eigen(system.Positions);
116         Eigen::VectorXf v0 = glm2eigen(system.Velocities);
117         Eigen::VectorXf f0_extern_grav = glm2eigen(std::vector<glm::vec3>(system.Positions.
    size(), glm::vec3(0, -system.Gravity, 0)));
118         Eigen::VectorXf system_m_vectorxf = glm2eigen(std::vector<glm::vec3>(system.Positions.
    size(), glm::vec3(system.Mass,system.Mass,system.Mass)));
119
120         Eigen::VectorXf y0 = x0 + ddt * (v0 + ddt * f0_extern_grav);
121
122
123         Eigen::VectorXf  grad_g = eval_grad_g(system, x0, y0, ddt);
124         Eigen::SparseMatrix<float> grad2_g(3 * pos_num, 3 * pos_num);
125         eval_grad2_g(system, x0, ddt, grad2_g);
```

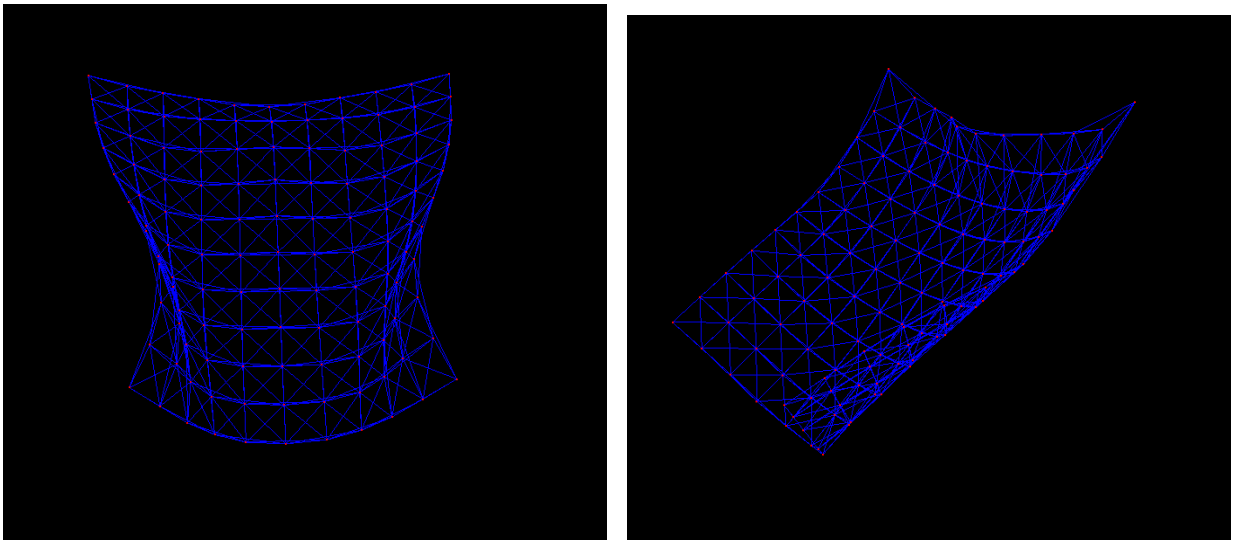
```
126
127         Eigen::VectorXf x1_x0 = ComputeSimplicialLLT(grad2_g, -grad_g);
128
129
130         Eigen::VectorXf  final_x    = x1_x0 + x0;
131         std::vector<glm::vec3> final_x_vec3 = eigen2glm(final_x);
132         std::vector<glm::vec3> final_v = eigen2glm(v0 + ddt * (eval_intern_f(system,final_x)/
    system.Mass + f0_extern_grav));
133
134         for (std::size_t i = 0; i < system.Positions.size(); i++) {
135             if (system.Fixed[i]) continue;
136             system.Velocities[i] = final_v[i];
137             system.Positions[i]  = final_x_vec3[i];
138         }
139
140     }
141
142
143 }
```

得到效果图：



(a) Implicit Euler-0　　(b) Implicit Euler-1

图 9: Task 2: Mass-Spring System-Implicit Euler



(a) Implicit Euler-2 (b) Implicit Euler-3

图 10: Task 2: Mass-Spring System-Implicit Euler