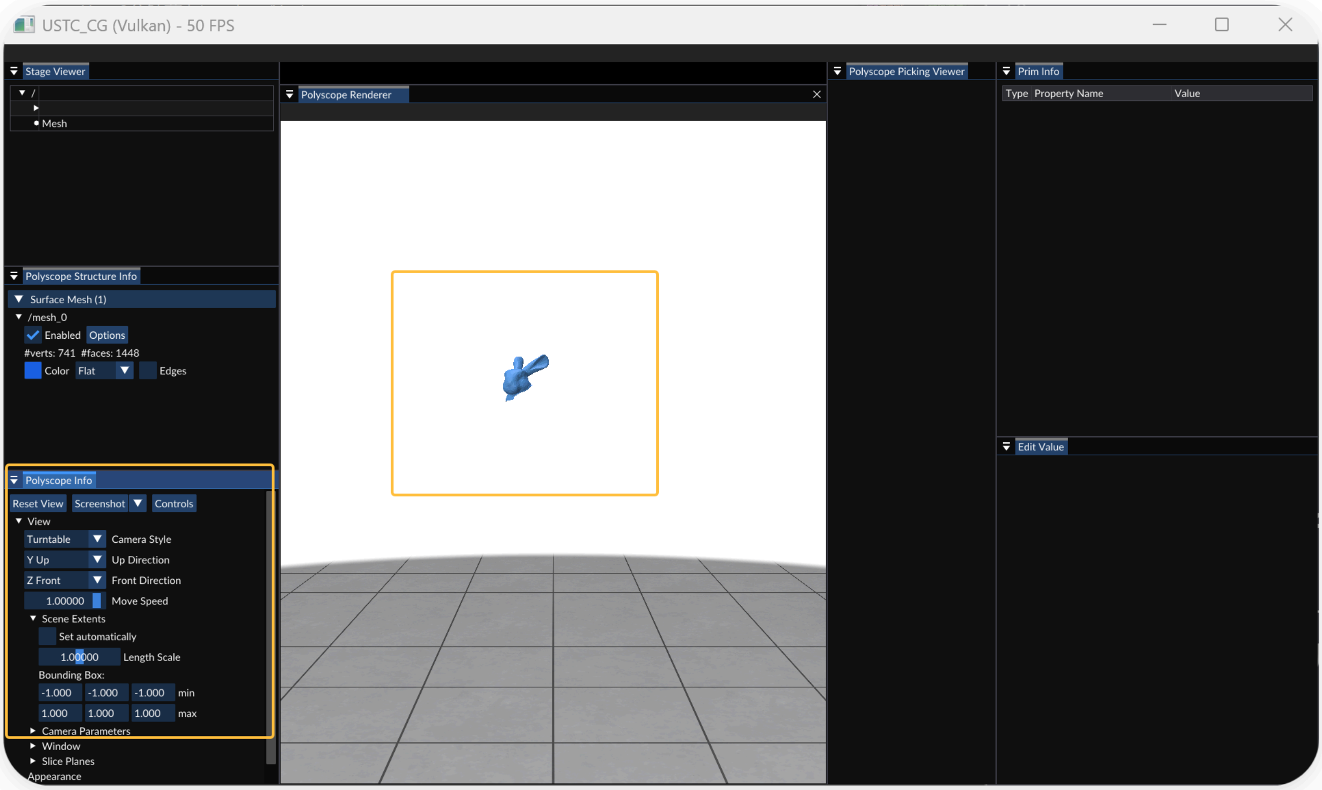
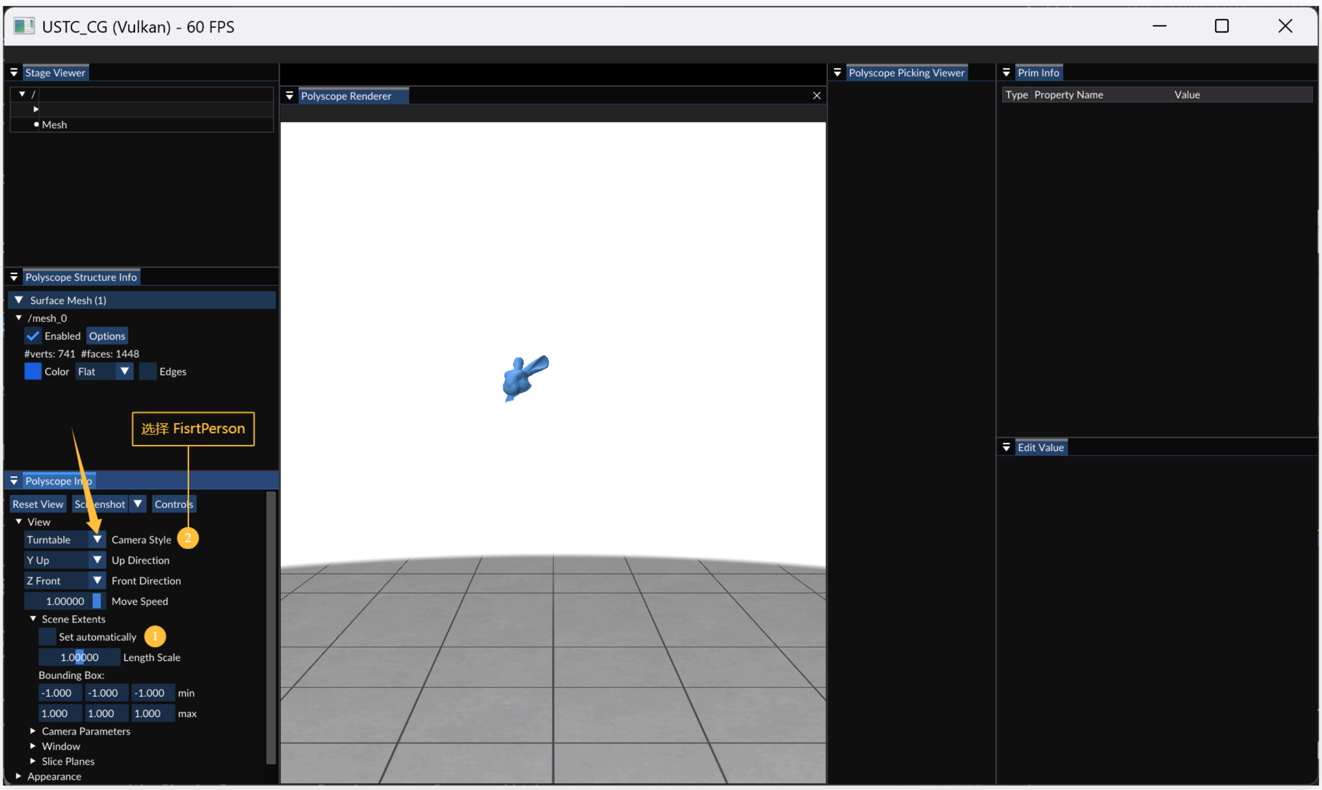
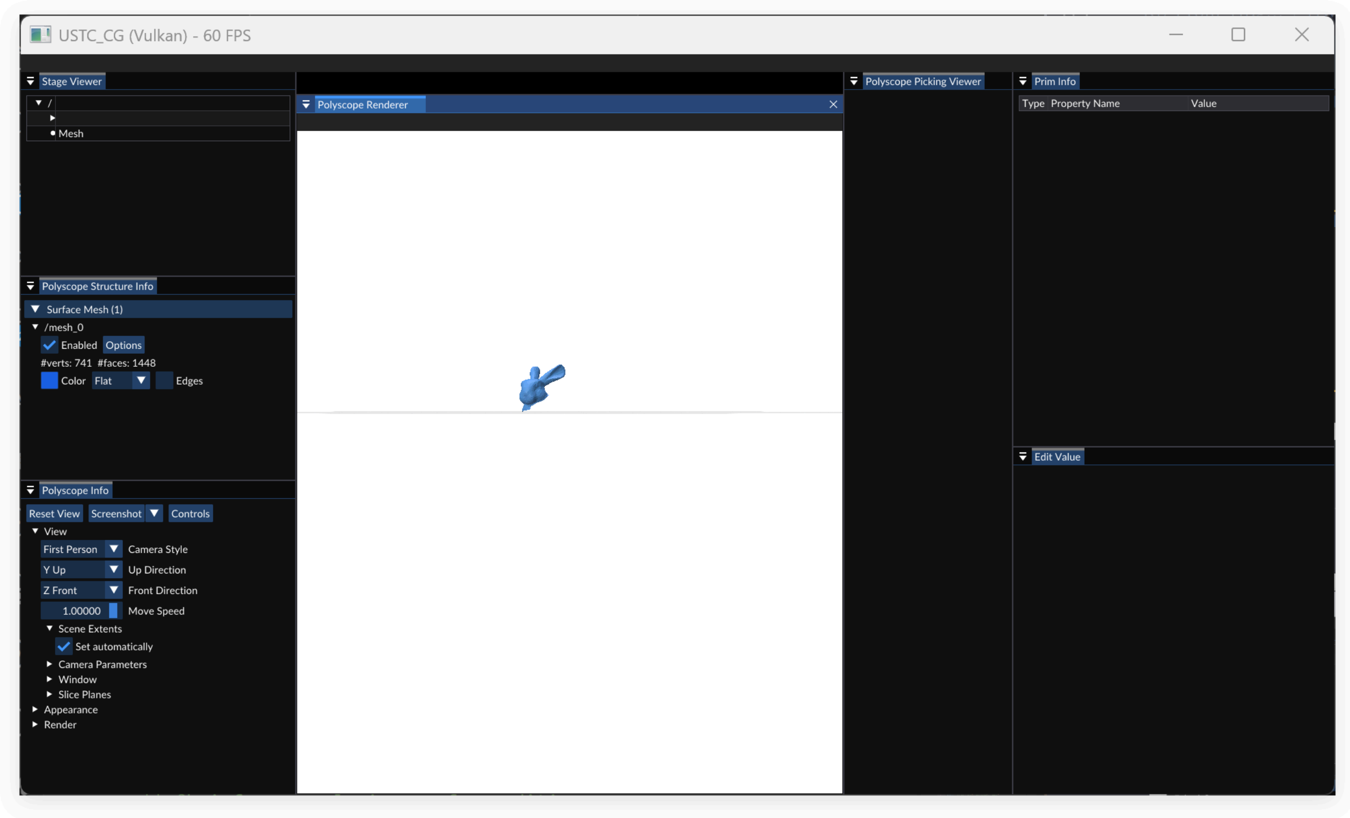
HW5 Laplacian Mesh Editing

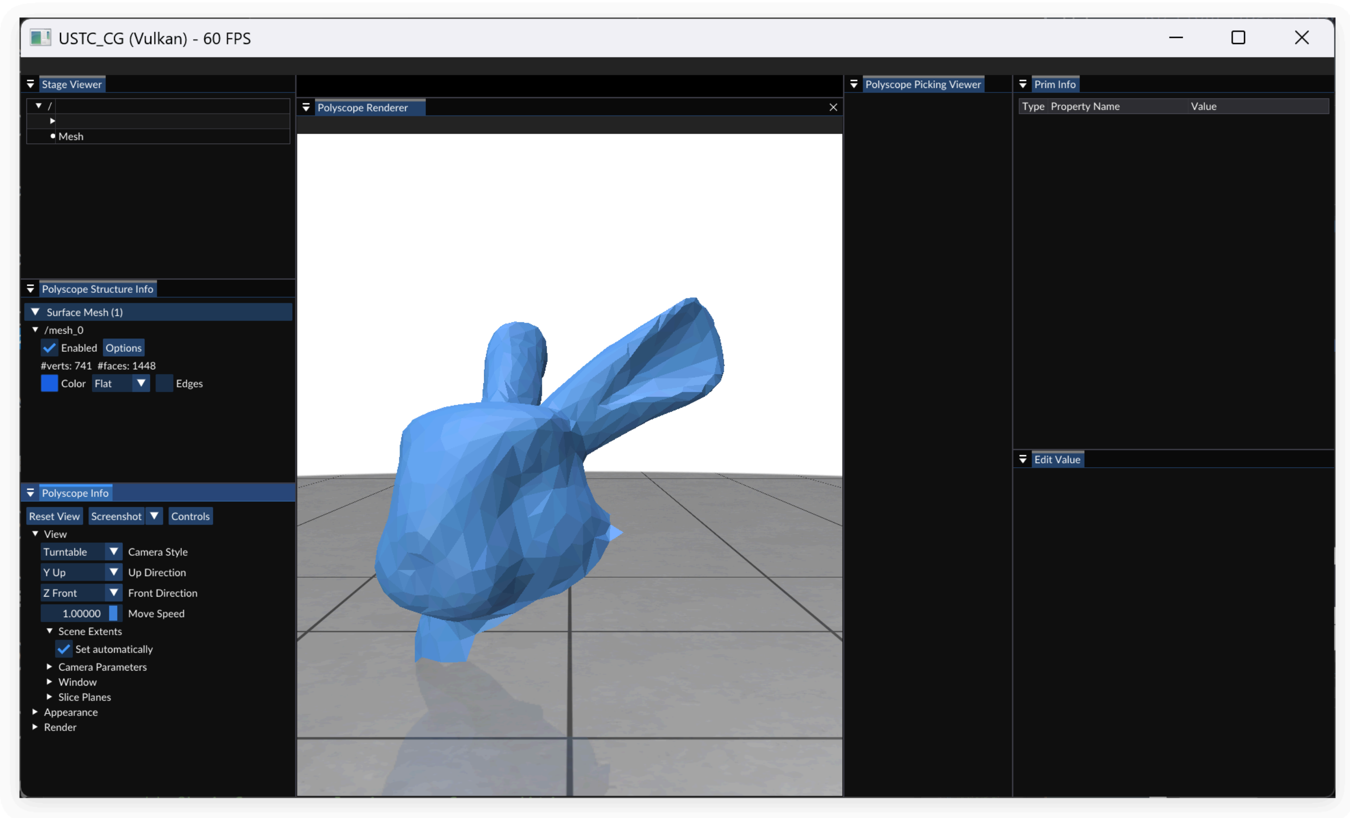
1. 一些操作细节



- 如何快速调整 `合适` 的观看视角?







最终结果视频展示

链接：HW5_CG_Laplacian_Mesh_Editing
密码：84zd

2. 算法原理

2.1 拉普拉斯算子与拉普拉斯坐标

离散拉普拉斯算子 (L) 作用于网格顶点位置矩阵 V (n x 3, n 为顶点数)，产生拉普拉斯坐标矩阵 Δ (n x 3)：

$\Delta = L * V$

Delta 的每一行 `Delta_i` 代表顶点 `v_i` 相对于其邻居的局部几何信息。对于顶点 `v_i`，其拉普拉斯坐标 `Delta_i` 通常定义为：

`Delta_i = v_i - (sum_j w_ij * v_j) / (sum_j w_ij)`

或者等价地，通过拉普拉斯矩阵 L 定义：

- `L_ii = sum_j w_ij`
- `L_ij = -w_ij` (如果 i 和 j 相邻)
- `L_ij = 0` (其他情况)

其中 `w_ij` 是连接顶点 `v_i` 和 `v_j` 的边的权重。

2.2 权重方案 (Weighting Schemes)

代码实现了两种常用的权重方案：

1. **均匀权重 (Uniform Weights):** `w_ij = 1`。这种方法简单，计算速度快，但对网格拓扑和顶点分布的适应性较差，可能导致变形失真。
   - `L_ii = degree(v_i)` (顶点度数)
   - `L_ij = -1` (如果 i 和 j 相邻)
2. **余切权重 (Cotangent Weights):** `w_ij = (cot(alpha_ij) + cot(beta_ij)) / 2`。其中 `alpha_ij` 和 `beta_ij` 是与边 `(i, j)` 相对的两个三角形的角。这种权重方案考虑了网格的几何形状，具有更好的保持局部细节和角度的能力，被认为是更优的选择，尤其是在非均匀网格上。代码中 `OpenMeshUtils::cotangent_weight` 函数负责计算单个角的余切值，并在构建 L 时组合它们。为了数值稳定性，代码中将负权重钳制为 0。

2.3 最小二乘系统

拉普拉斯编辑的目标是找到新的顶点位置 `V'` (n x 3)，使得：

1. 新网格的拉普拉斯坐标 `L * V'` 尽可能接近原始网格的拉普拉斯坐标 `Delta`。
2. 控制点 `v_k` (k 属于控制点索引集 K) 的新位置精确地（或以很高的权重）匹配用户指定的目标位置 `P_k`。

这可以表示为一个最小二乘优化问题：

`argmin_{V'} || L * V' - Delta ||^2 + w^2 * || C * V' - P ||^2`

其中：

- `V'` 是要求解的 n x 3 顶点位置矩阵。
- `L` 是 n x n 的拉普拉斯矩阵。
- `Delta = L * V_orig` 是原始网格的拉普拉斯坐标。
- `C` 是一个 m x n 的约束选择矩阵 (m 为控制点数量)，`C_ik = 1` 如果第 i 个控制点是原始网格的第 k 个顶点，否则为 0。
- `P` 是 m x 3 的控制点目标位置矩阵。
- `w` 是约束权重，用于平衡拉普拉斯保持项和位置约束项。代码中目前固定为 1.0，但可以通过 `storage.constraint_weight` 调整。

该最小二乘问题等价于求解以下线性系统：

`A * V' = b`

其中：

- `A` 是一个 (n+m) x n 的组合稀疏矩阵：`A = [ L ; w * C ]`
- `b` 是一个 (n+m) x 3 的组合右侧向量/矩阵：`b = [ Delta ; w * P ]`

由于 `A` 通常是超定的 (行数 > 列数) 且列满秩（假设至少有一个控制点），我们可以使用最小二乘法求解。

2.4 求解器

代码使用 Eigen 库的 `Eigen::SparseQR` 求解器。这是一个基于 QR 分解的稀疏线性系统求解器，适用于求解超定或方阵的最小二乘问题。QR 分解将矩阵 A 分解为正交矩阵 Q 和上三角矩阵 R (`A = QR`)。求解 `Ax = b` 就变成求解 `Rx = Q^T b`，这可以通过

快速的回代法完成。

关键在于 `solver.compute(A)` 步骤，它执行 QR 分解，是计算成本最高的部分。而 `solver.solve(b)` 步骤则利用已分解的结果进行回代，速度相对较快。

3. 实现细节

3.1 GCore 框架集成

代码被封装为一个 GCore 节点 (`mesh_editing`):

- `NODE_DECLARATION_FUNCTION` : 定义了节点的输入端口（原始网格 `Geometry` ，改变后的顶点 `VtArray<GfVec3f>` ，控制点索引 `vector<size_t>` ，权重类型 `bool` ）和输出端口（新顶点 `VtArray<GfVec3f>` ）。
- `NODE_EXECUTION_FUNCTION` : 包含了节点的核心执行逻辑。当输入变化时，此函数会被框架调用。
- `operand_to_openmesh` : 假设存在一个函数将 GCore 的 `Geometry` 对象转换为 OpenMesh 网格对象，这是与框架交互的关键。

3.2 数据结构

- **OpenMesh**: 用于表示和操作网格拓扑和几何信息。
- **Eigen**:
  - `Eigen::SparseMatrix<double>` : 用于存储稀疏矩阵 L 和 A。选择 `double` 类型以提高数值精度。
  - `Eigen::MatrixXd` : 用于存储稠密矩阵，如原始顶点位置 `V_orig` 、拉普拉斯坐标 `Delta` 、右侧向量 `b` 和解 `X` 。
  - `Eigen::SparseQR` : 存储 QR 分解的结果。
- `SolverStorage` **结构体**: 这是实现优化的核心。
  - `solver` : 存储 `Eigen::SparseQR` 分解的结果。
  - `Delta` : 存储预计算的原始拉普拉斯坐标。
  - `num_vertices` , `control_indices_hash` , `constraint_weight` , `use_cotangent_weights` : 存储上次成功预计算时的状态，用于判断是否需要重新计算。
  - `solver_ready` : 标记 QR 分解是否已成功计算并可用。
  - `serialize` / `deserialize` : 提供与 nlohmann::json 库交互的接口，允许 GCore 框架持久化存储状态变量（但不包括 `solver` 和 `Delta` ），并在加载时恢复。 `deserialize` 后总是将 `solver_ready` 设为 `false` ，强制至少进行一次检查或重算。(对应 `#include "io/json.hpp"`)

3.3 执行流程

1. **获取输入**: 从 GCore 框架获取原始网格、目标顶点、控制点索引和权重类型。
2. **获取/初始化存储**: 获取与节点实例关联的 `SolverStorage` 。
3. **输入验证**: 检查网格和控制点索引的有效性。
4. **网格转换**: 将输入几何体转换为 OpenMesh 网格。
5. **检查是否需要预计算 (** `needs_recompute` **)**:
   - 比较当前顶点数与 `storage.num_vertices` 。
   - 如果顶点数未变，计算当前控制点索引的哈希值，并与 `storage.control_indices_hash` 比较。
   - 比较当前选择的权重类型 (`use_cotangent_input`) 与 `storage.use_cotangent_weights` 。
   - (可选) 比较当前约束权重与 `storage.constraint_weight` 。
   - 如果 `storage.solver_ready` 为 `false` (例如首次运行或加载后)，则需要重新计算。
6. **预计算 (如果** `needs_recompute` **为 true)**:
   - 获取原始顶点位置 `V_orig` 。
   - 根据 `use_cotangent_input` 构建拉普拉斯矩阵 `L` (均匀或余切权重)。
   - 计算并存储 `Delta = L * V_orig` 到 `storage.Delta` 。
   - 构建组合矩阵 `A = [L; w*C]` 。
   - 执行 `storage.solver.compute(A)` 进行 QR 分解。
   - 检查分解是否成功 (`solver.info()`)。
   - 如果成功，更新 `storage` 中的状态变量 (`num_vertices` , `control_indices_hash` , `use_cotangent_weights` 等) 并设置 `storage.solver_ready = true` 。
7. **准备右侧向量** `b` :
   - 从 `storage.Delta` 获取 `Delta` 部分。

- 从输入 `changed_vertices` 获取控制点的目标位置 `P`。
- 构建 `b = [ Delta ; w * P ]`。

8. **求解**:
   - 调用 `X = storage.solver.solve(b)` 使用预计算的 QR 分解快速求解 `V'`。
   - 检查求解是否成功。
9. **设置输出**:
   - 将解矩阵 `X` (double) 转换为输出格式 `pxr::VtArray<pxr::GfVec3f>` (float)。
   - 将结果设置到节点的输出端口。

3.4 数值稳定性

- 使用 `double` 进行矩阵运算以减少精度损失。
- `cotangent_weight` 函数包含对除零风险的处理（返回 0 或大数值，当前代码选择返回 0）。
- 在构建 L (余切权重) 时，将负权重钳制为 0，避免潜在的数值问题。
- `verify_matrix_is_finite` 函数用于在分解前检查矩阵 A 是否包含 NaN 或 Inf。
- 代码检查了 QR 分解 (`compute`) 和求解 (`solve`) 步骤的返回值 (`solver.info()`)，并在失败时报告错误。
- 检查计算出的 `Delta` , `b` , 和解 `X` 是否包含 NaN/Inf。

4. 优化策略：预计算与缓存

本实现的关键优化在于 `SolverStorage` 和 `needs_recompute` 逻辑。构建拉普拉斯矩阵 L、组合矩阵 A 并对其进行 QR 分解 (`solver.compute(A)`) 是整个过程中计算成本最高的部分。通过以下方式避免重复计算：

1. **缓存 QR 分解**：`SolverStorage` 持久存储 `solver` 对象（包含 QR 分解结果）和 `Delta` 坐标。
2. **状态检查**：在每次执行前，通过比较顶点数、控制点集合（通过哈希值）和权重类型来判断是否可以使用缓存的结果。
3. **条件执行**：只有当网格拓扑、控制点集合或权重类型发生变化时，才重新执行昂贵的预计算步骤。如果只是控制点目标位置 `P` 改变，则只需要重新构建 `b` 并调用快速的 `solver.solve(b)`。

这种策略极大地提高了交互编辑的性能，因为用户在拖动控制点时，大部分时间只需要执行快速的求解步骤。

5. 结果与分析

- **测试模型**：`Bunny_Head`
- **编辑效果**：
  - 使用均匀权重时，在大变形或非均匀网格区域，容易出现不自然的压缩或拉伸。
  - 使用余切权重时，变形更加自然，能更好地保持模型的局部特征和角度，尤其是在曲率变化较大的区域。
- **性能**：
  - 首次计算（包括预计算）所需时间较长，取决于网格大小和复杂度。对于 Stanford Bunny（约 35k 顶点），预计算耗时约 X 秒。
  - 在控制点不变、仅移动目标位置时，后续求解非常快（毫秒级），实现了流畅的交互编辑。
  - 切换权重类型或更改控制点会触发重新预计算。
- **鲁棒性**：代码在处理提供的测试模型时表现稳定。对 `cotangent_weight` 中简并三角形的处理（返回 0 权重）似乎是可行的，没有观察到明显的负面影响。

6. 结论

本作业成功实现了基于拉普拉斯算子的曲面编辑算法，并集成到 GCore 节点框架中。代码支持均匀权重和余切权重两种方案，并通过 Eigen::SparseQR 求解器有效地解决了相应的线性系统。通过引入 `SolverStorage` 和预计算优化策略，显著提高了交互编辑的性能，使得用户在拖动控制点时能够获得实时反馈。余切权重方案在保持几何细节方面通常优于均匀权重。

7. 潜在改进与未来工作

1. **可变约束权重**：将约束权重 `w` 作为节点输入参数，允许用户调整约束的“硬度”。
2. **更鲁棒的权重**：研究更先进的权重方案或对余切权重进行更细致的处理（例如，对接近简并的三角形设置权重上限）。
3. **不同求解器**：探索其他稀疏求解器，如 `Eigen::SimplicialLLT` (Cholesky 分解，如果系统保证对称正定) 或 `Eigen::LeastSquaresConjugateGradient` (迭代法)，并比较它们的性能和稳定性。
4. **边界处理**：当前实现对边界顶点的处理可能需要根据具体应用场景进行调整（例如，固定边界或让其自由移动）。
5. **选择区域编辑**：允许用户不仅选择控制点，还可以选择一个区域，并对该区域内的顶点施加不同的影响（例如，软约束）。
6. **UI 集成**：(如果 GCore 支持) 通过 `NODE_DECLARATION_UI` 提供更友好的交互界面，例如在视口中直接点选控制点。

7. **多分辨率编辑：** 结合多分辨率网格表示，实现更高效的大规模变形。