HW4: report

> 姓名：杜朋澈
>
> ID：68
>
> 学号：PB21050988

算法

极小曲面计算

uniform weight

固定边界点坐标，取均匀权重下的 $\boldsymbol{\delta}_i = \mathbf{0}$ 即

$$\frac{1}{d_i}\sum_{j\in N(i)}(\boldsymbol{v}_i - \boldsymbol{v}_j) = \mathbf{0}, \quad \text{for all interior } i.$$

cotangent weight

$$w_j = \cot\alpha_{ij} + \cot\beta_{ij}$$

Tutte 参数化计算

分布边界点的坐标到平面凸区域的边界，求解同样的方程组：

$$\boldsymbol{v_i} - \sum_{j\in N(i)} w_j \boldsymbol{v_j} = \mathbf{0}, \quad \text{for all interior } i.$$

实现

algorithm lib

- `solve_transform`用于填充b以及分三个维度求解极小曲面坐标结果。考虑到对于不同权重下的极小曲面计算，只有A矩阵的填充方式是不同的。故封装如下方法：

```
void solve_transform(
    const Eigen::SparseMatrix<double>& A,
    int vertex_num,
    std::shared_ptr<PolyMesh> halfedge_mesh)
{
    //分维度求解
    for (int dim = 0; dim < 3; ++dim) {
        Eigen::SparseVector<double> b(vertex_num);

        //填充b
        for (const auto& vertex_handle : halfedge_mesh->vertices()) {
            if (vertex_handle.is_boundary()) {
                b.coeffRef(vertex_handle.idx()) = halfedge_mesh->point(vertex_handle)[dim];
            }
        }

        Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(A);
        solver.factorize(A);

        if (solver.info() != Eigen::Success) {
            throw std::runtime_error("Minimal Surface: Matrix A factorize failed.");
        }

        //求解
        Eigen::VectorXd x = solver.solve(b);

        //写回结果
        for (const auto& vertex_handle : halfedge_mesh->vertices()) {
            if (!vertex_handle.is_boundary()) {
                auto point = halfedge_mesh->point(vertex_handle);
                point[dim] = x(vertex_handle.idx());
                halfedge_mesh->set_point(vertex_handle, point);
            }
        }
    }
}
```

- `get_boundary_edges`用于获取边界半边索引数组，在边界映射中获取网格边界的方式是固定的，故将该方法封装：

```cpp
std::vector<int> get_boundary_edges(std::shared_ptr<PolyMesh> halfedge_mesh)
{
    std::vector<int> boundary_halfedges;
    //先获取首个边界半边
    for (const auto& halfedge_handle : halfedge_mesh->halfedges()) {
        if (halfedge_handle.is_boundary()) {
            boundary_halfedges.push_back(halfedge_handle.idx());
            break;
        }
    }
    if (boundary_halfedges.empty()) {
        throw std::runtime_error("No boundary edges.");
    }

    //随后根据半边数据结构特性顺序遍历所有边界半边即可
    int index = boundary_halfedges[0];
    do {
        auto this_handle = halfedge_mesh->halfedge_handle(index);
        int next_index = halfedge_mesh->next_halfedge_handle(this_handle).idx();
        boundary_halfedges.push_back(next_index);
        index = next_index;
    } while (index != boundary_halfedges[0]);

    // 最后一次循环会将起始点重复填入结果，故弹出一次
    boundary_halfedges.pop_back();

    return boundary_halfedges;
}
```

nodes

- min_surf node

  由于后续求解与坐标填充过程相同，故此处仅给出A的填充方式

```cpp
// uniform weight -----------------------------------------------------------
    int vertex_num = halfedge_mesh->n_vertices();
    Eigen::SparseMatrix<double> A(vertex_num, vertex_num);
    for (const auto& vertex_handle : halfedge_mesh->vertices()) {
        int idx = vertex_handle.idx();
        if (vertex_handle.is_boundary()) {
            A.coeffRef(idx, idx) = 1;
        }
        else {
            int neighbor_num = 0;
```

```
            for (const auto& out_halfedge : vertex_handle.outgoing_halfedges()) {
                ++neighbor_num;
                int neighbor_idx = out_halfedge.to().idx();
                // 所有邻居的权重均为1
                A.coeffRef(idx, neighbor_idx) = -1;
            }
            A.coeffRef(idx, idx) = neighbor_num;
        }
    }
    A.makeCompressed();
    solve_transform(A, vertex_num, halfedge_mesh);




// cotangent weight ---------------------------------------------------------------
    int vertex_num = halfedge_mesh->n_vertices();
    Eigen::SparseMatrix<double> A(vertex_num, vertex_num);
    for (const auto& vertex_handle : halfedge_mesh->vertices()) {
        int idx = vertex_handle.idx();
        //边界行与均匀权重相同
        if (vertex_handle.is_boundary()) {
            A.coeffRef(idx, idx) = 1;
        }
        else {
            double sum_weight = 0.0;
            for (const auto& out_halfedge : vertex_handle.outgoing_halfedges()) {
                double weight = 0.0;

                //获取neighbor，以及与neighbor和self共面的两个点
                int neighbor_idx = out_halfedge.to().idx();
                auto vi_idx = out_halfedge.prev().from().idx();
                auto vj_idx = out_halfedge.opp().next().to().idx();

                //获取这些点的位置即可计算向量夹角，进而计算权重
                auto pos_self = origin_mesh->point(origin_mesh->vertex_handle(idx));
                auto pos_neighbor = origin_mesh->point(origin_mesh->vertex_handle(neighbor_idx));
                auto i_pos = origin_mesh->point(origin_mesh->vertex_handle(vi_idx));
                auto j_pos = origin_mesh->point(origin_mesh->vertex_handle(vj_idx));

                auto vec_1_1 = pos_self - i_pos;
                auto vec_1_2 = pos_neighbor - i_pos;
                auto vec_2_1 = pos_self - j_pos;
                auto vec_2_2 = pos_neighbor - j_pos;
```

```cpp

                auto cos_theta_1 = vec_1_1.dot(vec_1_2) / (vec_1_1.norm() * vec_1_2.norm());
                auto cos_theta_2 = vec_2_1.dot(vec_2_2) / (vec_2_1.norm() * vec_2_2.norm());

                auto cot_theta_1 = cos_theta_1 / (std::sqrt(1 - cos_theta_1 * cos_theta_1));
                auto cot_theta_2 = cos_theta_2 / (std::sqrt(1 - cos_theta_2 * cos_theta_2));

                weight = cot_theta_1 + cot_theta_2;

                A.coeffRef(idx, neighbor_idx) = -weight;
                sum_weight += weight;
            }
            A.coeffRef(idx, idx) = sum_weight;
        }
    }
    A.makeCompressed();
    solve_transform(A, vertex_num, halfedge_mesh);

```

- mapping node

  由于获取边界半边索引的过程是相同的，故此处仅给出将边界映射到指定形状的代码：

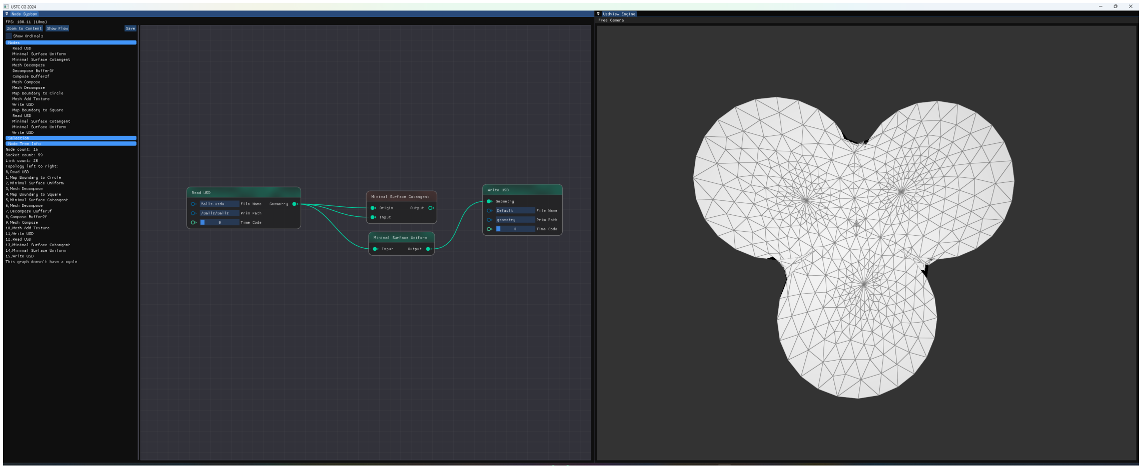
```cpp
// circle mapping -----------------------------------------------------------
    std::vector<int> boundary_halfedges = get_boundary_edges(halfedge_mesh);

    for (int i = 0; i < boundary_halfedges.size(); ++i) {
        auto bhe_idx = boundary_halfedges[i];
        auto halfedge_handle = halfedge_mesh->halfedge_handle(bhe_idx);
        auto vertex_handle = halfedge_mesh->to_vertex_handle(halfedge_handle);
        auto location = halfedge_mesh->point(vertex_handle);
        location[0] = std::cos(2.0 * M_PI * i / boundary_halfedges.size()) / 2 + 0.5;
        location[1] = std::sin(2.0 * M_PI * i / boundary_halfedges.size()) / 2 + 0.5;
        location[2] = 0.0;

        halfedge_mesh->set_point(vertex_handle, location);
    }




// square mapping -----------------------------------------------------------
```

```
    std::vector<int> boundary_halfedges = get_boundary_edges(halfedge_mesh);

    for (int k = 0; k < 4; k++) {
        double m = boundary_halfedges.size() / 4;
        for (int i = k * m; i < (k + 1) * m; ++i) {
            auto bhe_idx = boundary_halfedges[i];
            auto halfedge_handle = halfedge_mesh->halfedge_handle(bhe_idx);
            auto vertex_handle = halfedge_mesh->to_vertex_handle(halfedge_handle);
            auto location = halfedge_mesh->point(vertex_handle);
            switch (k) {
                case 0: {
                    location[0] = (i - k * m) / m;
                    location[1] = 0;
                    break;
                }
                case 1: {
                    location[0] = 1;
                    location[1] = (i - k * m) / m;
                    break;
                }
                case 2: {
                    location[0] = 1 - ((i - k * m) / m);
                    location[1] = 1;
                    break;
                }
                case 3: {
                    location[0] = 0;
                    location[1] = 1 - ((i - k * m) / m);
                    break;
                }
            }
            location[2] = 0;
            halfedge_mesh->set_point(vertex_handle, location);
        }
    }

```

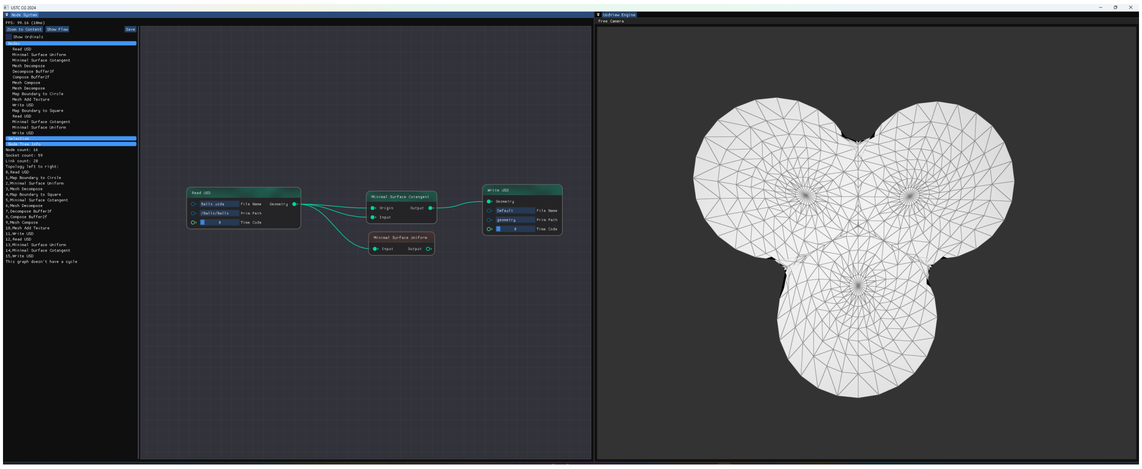
演示

最小曲面

- uniform weight (Balls)

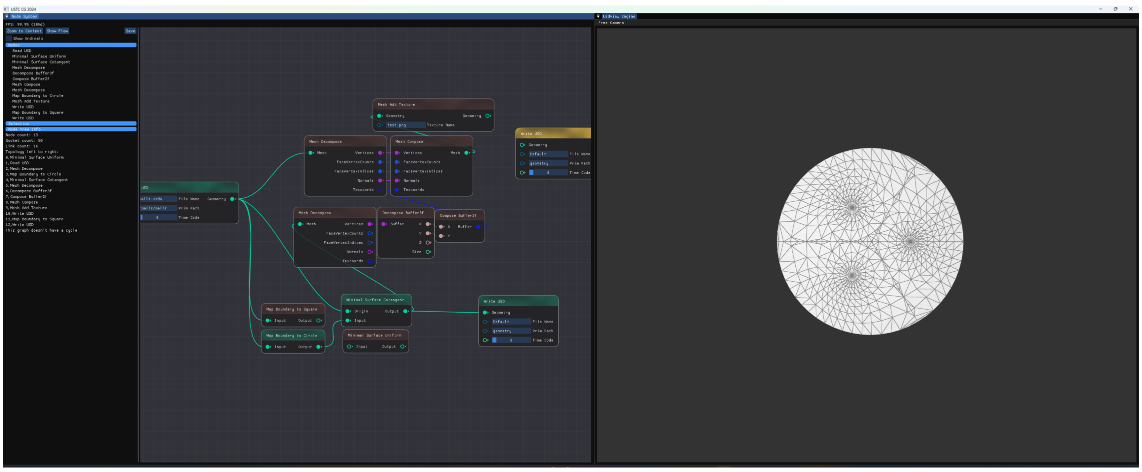


- cotangent weight (Balls)

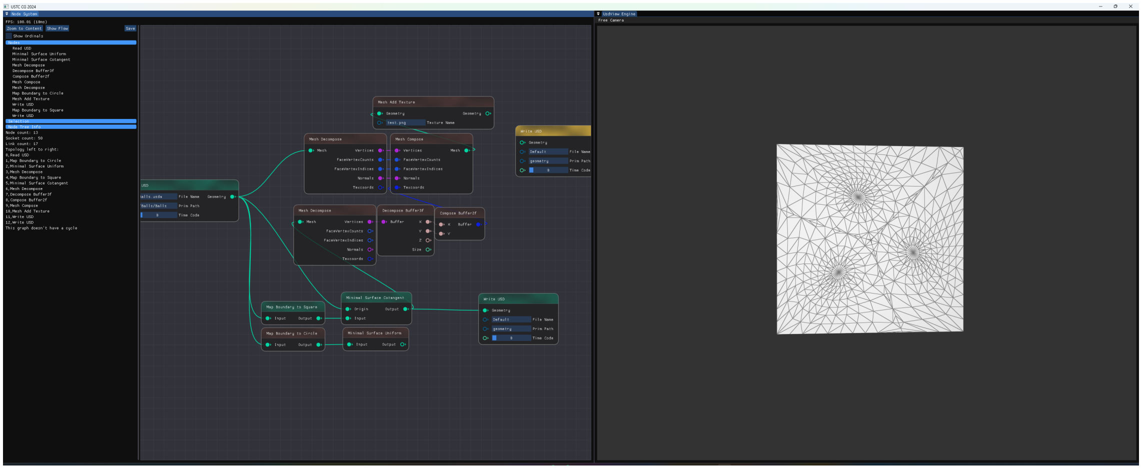


边界映射

- circle (Balls, cotangent weight)

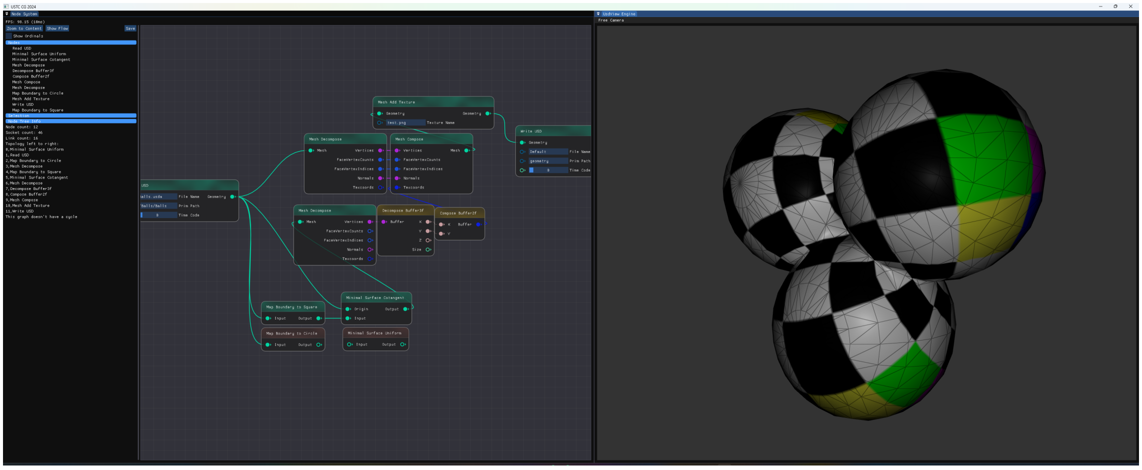


- square (Balls, cotangent weight)



参数化

- Balls (square map, cotangent weight)



- CatHead (square map, cotangent weight)