Lab 2 报告

蒋锦豪 2400011785

2025 年 10 月 26 日

Task 1: Loop Mesh Subdivision

实现方法 按照下面的步骤进行.

更新原有顶点: 遍历 `prev_mesh` 中的所有顶点 `v`, 按照下面的方法计算新的位置 `curr_v_pos`:

$$\boldsymbol{v}^* = (1 - nu)\boldsymbol{v} + \sum_{i=1}^{n} u\boldsymbol{v}_i, \quad u = \begin{cases} 3/16, & n = 3 \\ 3/(8n), & n > 3 \end{cases}$$

其中 n 为顶点邻接点的数目, 根据 `v` 的邻居集合 `neighbors` 的长度 `n` 可以得到; 然后根据 `n` 计算 `u`, 最后求和后将结果存入 `curr_mesh.Positions` 中即可. 循环体内的代码如下:

```
DCEL::VertexProxy const * v         = G.Vertex(i);
std::vector<uint32_t>     neighbors = v->Neighbors();
// your code here:
glm::vec3   curr_v_pos { 0.0f };
std::size_t n = neighbors.size();
float       u = (n == 3) ? (3.0f / 16.0f) : (3.0f / (8.0f * n));
curr_v_pos += (1.0f - u * n) * prev_mesh.Positions[i];
for (std::size_t j = 0; j < n; ++j) curr_v_pos += u * prev_mesh.Positions[neighbors[j]];
curr_mesh.Positions.push_back(curr_v_pos);
```

增设新顶点: 遍历 `prev_mesh` 中的所有边 `e`, 对于每一条边, 如果它没有对偶半边, 则说明它是边界边, 新顶点的位置即边的终点, 代码如下:

```
std::size_t start     = e->From();
std::size_t end       = e->To();
glm::vec3   new_v_pos = 0.5f * prev_mesh.Positions[start] + 0.5f * prev_mesh.Positions[end];
curr_mesh.Positions.push_back(new_v_pos);
```

如果它有对偶半边, 则按照下面的公式计算新顶点的位置:

$$\boldsymbol{v}^* = \frac{3}{8}(\boldsymbol{v}_1 + \boldsymbol{v}_2) + \frac{1}{8}(\boldsymbol{v}_3 + \boldsymbol{v}_4)$$

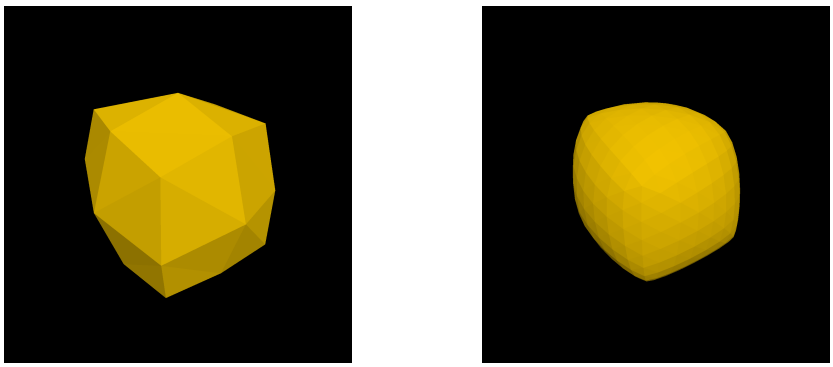
其中 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 是边的两个端点,$\boldsymbol{v}_3, \boldsymbol{v}_4$ 是与该边 (及其对偶边) 相对的顶点, 可以用 `e->OppositeVertex()` 获取. 代码如下:

```
std::size_t start      = e->From();
std::size_t end        = e->To();
std::size_t opposite_1 = e->OppositeVertex();
std::size_t opposite_2 = eTwin->OppositeVertex();
glm::vec3   new_v_pos  = 0.375f * (prev_mesh.Positions[start] + prev_mesh.Positions[end])
    + 0.125f * (prev_mesh.Positions[opposite_1] + prev_mesh.Positions[opposite_2]);
curr_mesh.Positions.push_back(new_v_pos);
```

最后按照原有顶点和新增设顶点的几何关系, 将新的三角形面片加入 `curr_mesh` 中. 代码如下:

```
std::uint32_t toInsert[4][3] = {
    // your code here:
    { v0, m2, m1 }, { v1, m0, m2 },
    { v2, m1, m0 }, { m0, m1, m2 }
};
```

实现效果 对立方体 `cube.obj` 分别进行一次和三次细分的结果如下:

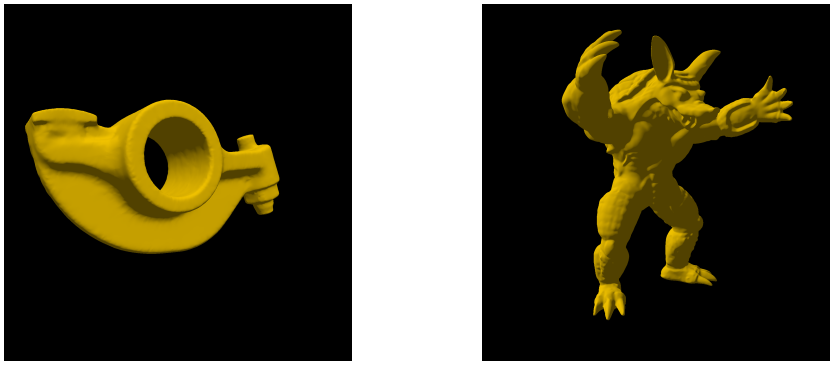


(a) 迭代次数 = 1　　(b) 迭代次数 = 3

图 1: 对 `cube.obj` 进行 Loop Mesh Subdivision 的结果

对其他模型进行三次细分的结果如下:

(a) 模型 block.obj　(b) 模型 dinosaur.obj　(c) 模型 fandisk.obj



(d) 模型 rocker.obj　(e) 模型 arma.obj

图 2: 对其他模型进行 Loop Mesh Subdivision 的结果

Task 2: Spring-Mass Mesh Parameterization

实现方法　按照下面的步骤进行.

首先需要遍历图形上的顶点, 用 `v->OnBoundary()` 方法找到一个边界上的点, 其序号保存为 `start`.

```
std::size_t start = 0;
DCEL::VertexProxy const * v = G.Vertex(start);
while (! v->OnBoundary() && start < input.Positions.size()) {
    start++;
    DCEL::VertexProxy const * v = G.Vertex(start);
} start--;
```

从 `start` 出发, 沿着边界遍历图形, 将边界上顶点的序号按顺序存入 `boundary` 中. 然后将剩余点 (即图形内部的点) 的序号存入 `inside` 中, 并建立一个映射 `inside_index`, 将原图形中的顶点序号映射到 `inside` 中的序号, 方便后面线性方程的求解. 代码如下:

```
std::vector<std::size_t> boundary, inside, inside_index(input.Positions.size(), 0);
boundary.push_back(start);
```

```
while (1) {
    DCEL::VertexProxy const * v = G.Vertex(boundary.back());
    std::pair<std::uint32_t, std::uint32_t> boundary_v_neighbors = v->BoundaryNeighbors();
    if (boundary_v_neighbors.first == start) break;
    boundary.push_back(boundary_v_neighbors.first);
}
for (std::size_t i = 0; i < input.Positions.size(); ++i) {
    if (std::find(boundary.begin(), boundary.end(), i) == boundary.end()) {
        inside_index[i] = inside.size(); inside.push_back(i);
    }
}
```

现在将边界点映射到 $[0,1]\times[0,1]$ 的圆上. 首先将各个边界边的长度当作圆的弧长以计算边界的周长 `perimeter`. 用 `theta` 变量记录当前点在圆上的角度, 每次增加的角度为边的长度 `boundary_length[i]` 与周长 `perimeter` 的比值乘以 2π. 代码如下:

```
std::size_t        boundary_v_num = boundary.size(), inside_v_num = inside.size();
float              perimeter = glm::length(input.Positions[boundary[0]] - input.Positions[boundary.
    back()]);
std::vector<float> boundary_length;
for (std::size_t i = 1; i < boundary_v_num; ++i) {
    boundary_length.push_back(glm::length(input.Positions[boundary[i]] - input.Positions[boundary[i
    - 1]]));
    perimeter += boundary_length.back();
}
float theta             = 1.1f;
for (std::size_t i = 0; i < boundary_v_num; ++i) {
    output.TexCoords[boundary[i]] = {
        0.5f + 0.5f * std::cos(theta),
        0.5f + 0.5f * std::sin(theta),
    };
    theta += boundary_length[i] / perimeter * 2 * 3.14159265359;
}
```

对于内部点的 uv 坐标 $\boldsymbol{t}_i$, 需要求解线性方程组, 并将系数按照平均系数设置系数:

$$\boldsymbol{t}_i - \frac{1}{n}\sum_{\boldsymbol{t}_j\in\Omega(\boldsymbol{t}_i)}\boldsymbol{t}_j = \boldsymbol{0}$$

如果 $\boldsymbol{t}_j$ 是内部点, 需要将系数矩阵 `A` 的对应位置 (这个位置可以由 $\boldsymbol{t}_j$ 在 `inside_index` 中的索引得到) 设为 $-1/n$; 如果是边界点, 则需要将前面计算出的 uv 坐标乘以 $1/n$ 后加入到系数向量 `b` 中, 以此求解方程 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$. 代码如下:

```cpp
std::vector<std::vector<float>> A(inside_v_num, std::vector<float>(inside_v_num, 0.0f));
std::vector<glm::vec2>          x(inside_v_num, { 0.0f, 0.0f }), b(inside_v_num, { 0.0f, 0.0f });
for (std::size_t i = 0; i < inside_v_num; ++i) {
    DCEL::VertexProxy const * v                 = G.Vertex(inside[i]);
    std::vector<uint32_t>     inside_v_neighbors = v->Neighbors();
    std::size_t               n                 = inside_v_neighbors.size();
    A[i][i]                                     = 1.0f;
    for (auto j : inside_v_neighbors) {
        float                     lambda_v = 1.0f / n;
        DCEL::VertexProxy const * u        = G.Vertex(j);
        if (u->OnBoundary()) {
            b[i] += lambda_v * output.TexCoords[j];
        } else {
            A[i][inside_index[j]] = -lambda_v;
        }
    }
}
```

最后按照 Gauss-Seidel 方法求解线性方程组. 代码如下:

```cpp
for (int k = 0; k < numIterations; ++k) {
    // your code here:
    for (std::size_t i = 0; i < inside_v_num; ++i) {
        glm::vec2 delta { 0.0f };
        for (std::size_t j = 0; j < inside_v_num; ++j) {
            if (j != i) delta += A[i][j] * x[j];
        }
        x[i] = b[i] - delta;
    }
}
```

实现效果 迭代次数 $iter = 0, 50, 1000$ 的结果分别如下:

(a) 迭代次数 = 0　　(b) 迭代次数 = 50　　(c) 迭代次数 = 1000

图 3: 对人脸模型进行 Spring-Mass Mesh Parameterization 的结果

Task 3: Mesh Simplification

实现方法　按照下面的步骤进行. 由于计算精度要求比较高, 在程序中统一使用 `double` 类型的矩阵和向量进行运算.

首先计算面 f 的误差矩阵 $\boldsymbol{K}_p$. 文献中给出的定义为: 对于给定的面 f, 其平面方程为 $ax+by+cz+d=0$, 即向量 $\boldsymbol{p}=\begin{pmatrix}a & b & c & d\end{pmatrix}^{\mathrm{t}}$, f 的代价矩阵 $\boldsymbol{K}_p$ 定义为 $\boldsymbol{p}$ 的外积.

注意到上述平面的法向量 $\boldsymbol{n}$ 恰为 (a,b,c)(注意单位化), 这可以由三角形两边 $\boldsymbol{e}_1$ 和 $\boldsymbol{e}_2$ 的叉积得到. 而 d 可以由点法式平面方程 $d=-\boldsymbol{n}\cdot\boldsymbol{v}$ 得到, 其中 $\boldsymbol{v}$ 为三角形的一个顶点. 用 `glm` 库中的各种向量运算功能即可完成上述过程, 代码如下:

```
auto UpdateQ {
    [&G, &output](DCEL::Triangle const * f) -> glm::dmat4 {
        // your code here:
        glm::dvec3 v0 = output.Positions[f->VertexIndex(0)], v1 = output.Positions[f->VertexIndex(1)], v2 = output.Positions[f->VertexIndex(2)];
        glm::dvec3 e1 = v1 - v0, e2 = v2 - v0;
        glm::dvec3 n = glm::normalize(glm::cross(e1, e2));
        glm::dvec4 p = { n.x, n.y, n.z, -glm::dot(n, v0) };
        return glm::outerProduct(p, p);
    }
};
```

然后是 `MakePair` 函数的实现, 主要是对于合法的顶点对 $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ 求解最优坍缩位置 $\boldsymbol{v}$, 和代价 $\Delta(\boldsymbol{v})$. 首先考虑矩阵 $\boldsymbol{q}$ 定义为

$$\boldsymbol{q} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中 q_{ij} 是输入矩阵 $\boldsymbol{Q}$ 的元素 (这里注意在 `glm::mat4` 中矩阵元素是按列优先存储的, 修改元素时需要注意). 计算 $\boldsymbol{q}$ 的行列式判断其可逆性. 如果 $\boldsymbol{q}$ 可逆, 就按照

$$\boldsymbol{v} = \boldsymbol{q}^{-1} \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}^{\mathrm{t}}$$

计算得到最优坍缩位置 $\boldsymbol{v}$; 否则就将 $\boldsymbol{v}$ 设为 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 的中点. 最后, 坍缩位置的代价为二次型 (这里对于两种情况下, 计算代价都应该用输入的矩阵 $\boldsymbol{Q}$ 来计算, 而不是用 $\boldsymbol{q}$, 否则会导致错误):

$$\Delta(\boldsymbol{v}) = \boldsymbol{v}\boldsymbol{Q}\boldsymbol{v}^{\mathrm{t}}$$

最后将各要素合并并返回. 代码如下:

```
static constexpr auto MakePair {
    [](DCEL::HalfEdge const * edge,
       glm::dvec3 const &     p1,
       glm::dvec3 const &     p2,
       glm::dmat4 const &     Q) -> ContractionPair {
        // your code here:
        ContractionPair result {};
        result.edge = edge;
        glm::dmat4 q = Q;
        q[0][3] = 0; q[1][3] = 0; q[2][3] = 0; q[3][3] = 1;
        if (glm::determinant(q) >= 0.001 || glm::determinant(q) <= -0.001) {
            result.targetPosition = glm::inverse(q) * (glm::dvec4 { 0, 0, 0, 1 });
        } else { result.targetPosition = { 0.5 * (p1 + p2), 1.0 }; }
        result.cost = glm::dot(result.targetPosition, Q * result.targetPosition);
        return result;
    }
};
```

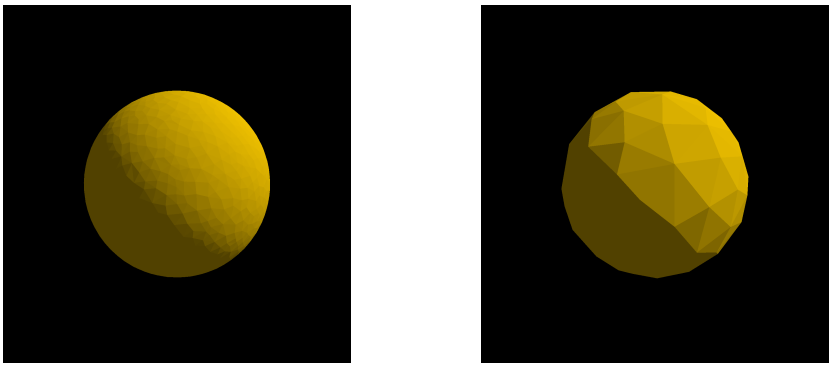
在主循环中实现坍缩后首先需要更新误差矩阵表 `Qv`. 遍历新顶点的环 `ring` 中的边 `e`, 那么所有面片即为 `f=e->Face()`. 对于 `e` 的起点和终点, 它们的误差矩阵中只有 `f` 贡献的部分需要修改, 新的量为 `UpdateQ(f)`, 原来的量为 `Kf[G.IndexOf(f)]`. 对于 `v1` 只需加上 `UpdateQ(f)` 即可. 全部修改完成后需要更新 `Kf` 中对应的元素. 代码如下:

```
for (auto e : ring) {
    // your code here:
    //     1. Compute the new Kp matrix for $e->Face()$.
    //     2. According to the difference between the old Kp (in $Kf$) and the new Kp (computed in step 1),
    //        update Q matrix of each vertex on the ring (update $Qv$).
    //     3. Update Q matrix of vertex v1 as well (update $Qv$).
    //     4. Update $Kf$.
    auto f      = e->Face(); auto Q      = UpdateQ(f);
    auto modify = Q - Kf[G.IndexOf(f)];
    Qv[e->From()] += modify; Qv[e->To()] += modify; Qv[v1] += Q;
    Kf[G.IndexOf(f)] = Q;
}
```

最后需要更新所有代价矩阵被修改的顶点 (即 `v1` 所有邻居顶点 `v`) 所连的边. 同样地, 遍历 `v` 周围的环 `ring_out` 上的边 `e_out`, 它的下一条边 `e_modify = e_out->PrevEdge()` 就是需要修改的边. 在修改时需要注意如果改动后不能保持空间拓扑性则需要删除该顶点对. 代码如下:

```
// Finally, as the Q matrix changed, we should update the relative $ContractionPair$ in $pairs$.
// Any pair with the Q matrix of its endpoints changed, should be remade by $MakePair$.
// your code here:
for (auto e : ring) {
    auto v        = e->From();
    auto ring_out = (G.Vertex(v))->Ring();
    for (auto e_out : ring_out) {
        auto v_out    = e_out->From();
        auto e_modify = e_out->PrevEdge();
        if (! G.IsContractable(e_modify)) {
            pairs[pair_map[G.IndexOf(e_modify)]].edge = nullptr;
        } else {
            pairs[pair_map[G.IndexOf(e_modify)]] = MakePair(e_modify, output.Positions[v], output.Positions[v_out], Qv[v] + Qv[v_out]);
        }
    }
}
```

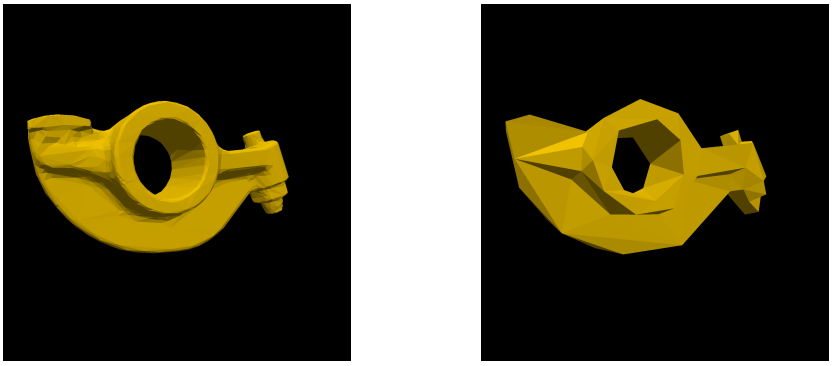
实现效果 对立方体 `sphere.obj` 进行简化的结果如下:



(a) 迭代次数 = 2　　(b) 迭代次数 = 4

图 4: 对 `sphere.obj` 进行 Mesh Simplification 的结果

对 `rocker.obj` 进行简化的结果如下:



(a) 迭代次数 = 2　　(b) 迭代次数 = 4

图 5: 对 `rocker.obj` 进行 Mesh Simplification 的结果

对其他模型进行简化的结果如下 (迭代次数 = 4):

(a) 模型 `fandisk.obj`　　(b) 模型 `arma.obj`

图 6: 对其他模型进行 Mesh Simplification 的结果

Task 4: Mesh Smoothing

实现方法 按照下面的步骤进行.

首先对于给定角 θ 的顶点 `vAngle` 和两边上的点 `v1,v2`, 按照下面的方法计算其余切值:

$$\cot\theta = \frac{\cos\theta}{\sin\theta} = \frac{\boldsymbol{e}_1 \cdot \boldsymbol{e}_2}{||\boldsymbol{e}_1 \times \boldsymbol{e}_2||}$$

其中 $\boldsymbol{e}_1, \boldsymbol{e}_2$ 分别为角的两边. 得到结果后, 为避免后续计算出现极端的数值, 需通过 `std::clamp` 函数将其限制在 $[0.1, 5]$ 内. 代码如下:

```
static constexpr auto GetCotangent {
    [](glm::vec3 vAngle, glm::vec3 v1, glm::vec3 v2) -> float {
        // your code here:
        glm::vec3   u         = v1 - vAngle;
        glm::vec3   v         = v2 - vAngle;
        float       cross_len = glm::length(glm::cross(u, v));
        return std::clamp(glm::dot(u, v) / cross_len, 0.1f, 5.0f);
    }
};
```

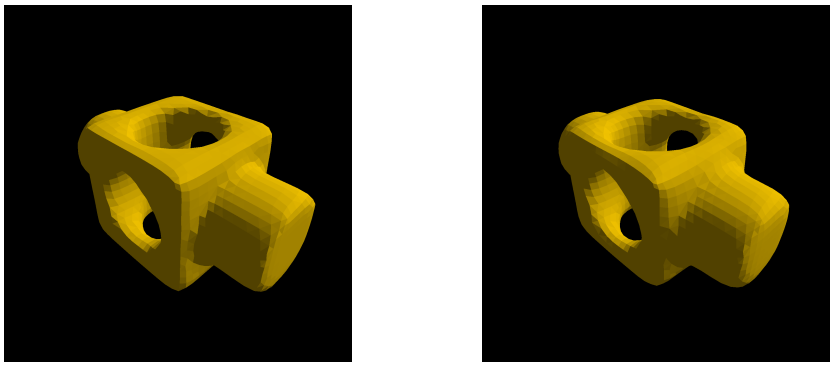
然后按照实现方法的不同分别进行新顶点的运算位置的运算, 代码如下:

```
for (std::uint32_t iter = 0; iter < numIterations; ++iter) {
    Engine::SurfaceMesh curr_mesh = prev_mesh;
    for (std::size_t i = 0; i < input.Positions.size(); ++i) {
        // your code here: curr_mesh.Positions[i] = ...
        auto v = G.Vertex(i);
        auto      neighbors = v->Neighbors();
        int       n         = neighbors.size();
        glm::vec3 sum { 0.0f };
        if (useUniformWeight) {
            for (std::size_t j = 0; j < n; ++j) sum += prev_mesh.Positions[neighbors[j]];
            sum = (1.0f / n) * sum;
        } else {
            auto ring = v->Ring();
            float sum_w = 0.0f;
            for (auto e_ring : ring) {
                auto e = e_ring->NextEdge();
                std::vector<glm::vec3> pos(4, glm::vec3 { 0.0f });
                pos[0] = prev_mesh.Positions[e->To()];
                pos[1] = prev_mesh.Positions[e->From()];
                pos[2] = prev_mesh.Positions[e->NextEdge()->To()];
```

```
                pos[3] = prev_mesh.Positions[e->TwinEdge()->PrevEdge()->From()];
                float w = GetCotangent(pos[2], pos[0], pos[1]) + GetCotangent(pos[3], pos[0], pos
    [1]);
                sum_w += w;
                sum += w * (pos[1]);
            }
            sum /= sum_w;
        }
        curr_mesh.Positions[i] = (1 - lambda) * prev_mesh.Positions[i] + lambda * sum;
    }
    // Move curr_mesh to prev_mesh.
    prev_mesh.Swap(curr_mesh);
}
```

实现效果 下面都采用 $\lambda = 0.5$.

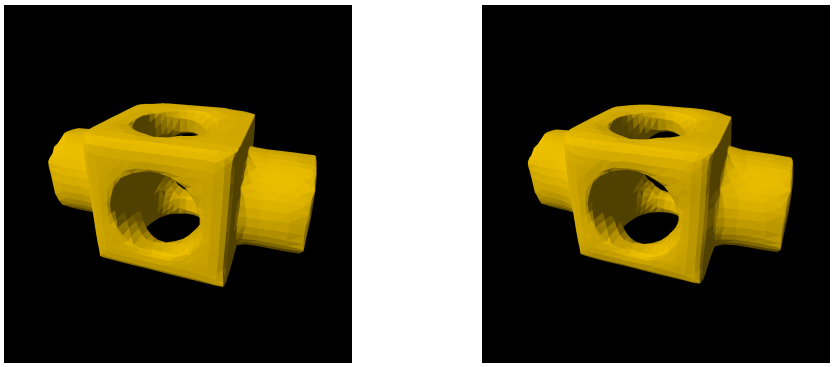
对 `block.obj` 进行 Uniform Laplacian 平滑的结果如下:



(a) 迭代次数 = 5　　(b) 迭代次数 = 10

图 7: 对 `block.obj` 进行 Uniform Laplacian Mesh Smoothing 的结果

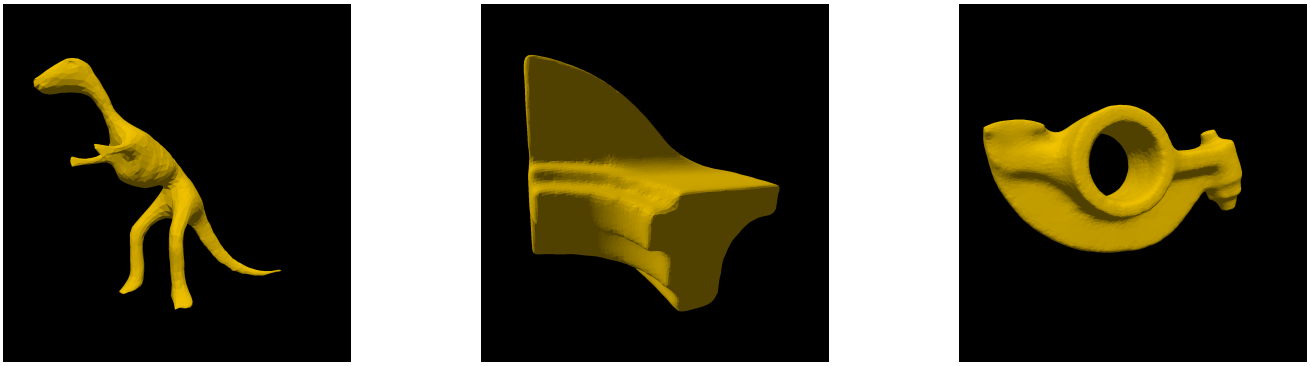
对 `block.obj` 进行 Cotangent Laplacian 平滑的结果如下:



(a) 迭代次数 = 5　　(b) 迭代次数 = 10

图 8: 对 `block.obj` 进行 Cotangent Laplacian Mesh Smoothing 的结果

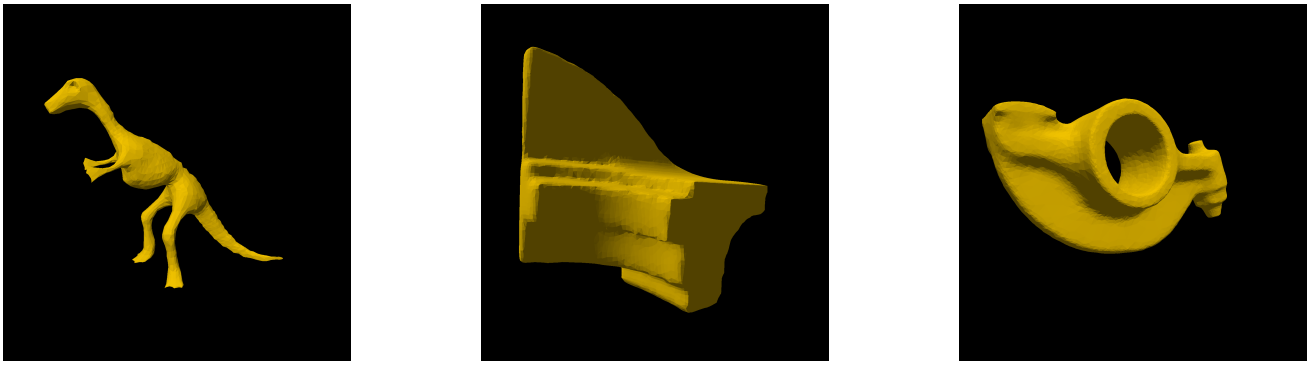
对其它图形进行 Uniform Laplacian 平滑的结果如下:



(a) 模型 `dinosaur.obj`, 迭代次数 = 10　(b) 模型 `fandisk.obj`, 迭代次数 = 10　(c) 模型 `rocker.obj`, 迭代次数 = 10

图 9: 对其他模型进行 Uniform Laplacian Mesh Smoothing 的结果

对其它图形进行 Cotangent Laplacian 平滑的结果如下:



(a) 模型 `dinosaur.obj`, 迭代次数 = 10　(b) 模型 `fandisk.obj`, 迭代次数 = 10　(c) 模型 `rocker.obj`, 迭代次数 = 10

图 10: 对其他模型进行 Cotangent Laplacian Mesh Smoothing 的结果

Task 5: Marching Cubes

实现方法 按照下面的步骤进行.

首先建立四维数组 `grid[x][y][z][w]` 来存储网格中以 (x, y, z)(这里是序号而非实际位置) 为起点的, 方向为 w(取值为 $0, 1, 2$, 含义与 `unit` 矩阵定义的方向向量相同) 的边上的点在 `output.Positions` 中的序号, 并将其初值赋值为 -1 以标记该点不在 Mesh 中.

然后逐个处理体素. 根据每个体素, 先计算各顶点的位置储存在 `v_pos` 中, 然后按照约定计算体素形式对应的二进制数 `v_hash`.

按照 `v_hash` 根据 `c_EdgeStateTable` 得到关于边上有无顶点的二进制数 `e_hash`, 然后根据 `e_hash` 判断第 j 条边 e_j 上是否有顶点. 如果有, 则根据线性插值计算该顶点的位置, 并查询 `grid` 数组中该位置是否已经存在该顶点, 如果不存在则将其加入 `output.Positions` 中, 并将其序号存入 `grid` 中, 然后将该顶点的序号存入 `idx_e` 中便于下一步的面的构造.

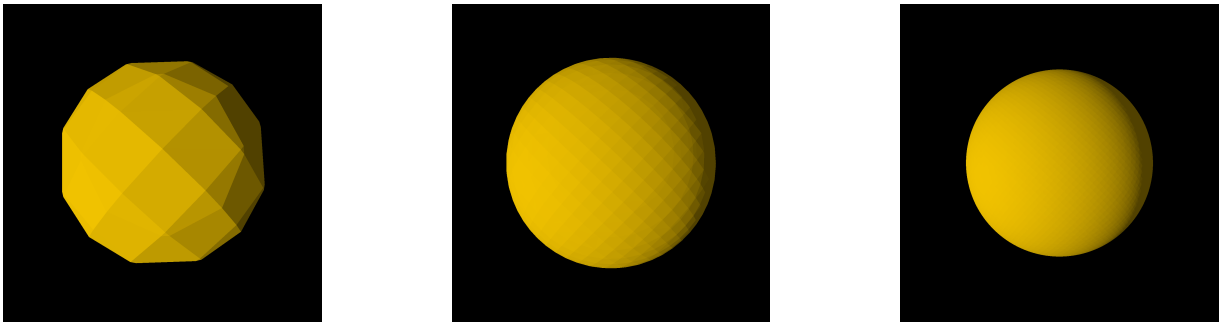
最后, 按照 `v_hash` 根据 `c_EdgeStateTable` 得到关于体素内三角形面的信息, 然后按照 `idx_e` 的索引将对应的边存入 `output.Indices` 中即可.

上述过程的代码如下 (只给出了循环体内的代码):

```
std::vector<glm::vec3> v_pos(8, glm::vec3 { 0.0f });
v_pos[0] = grid_min + glm::vec3 { nx * dx, ny * dx, nz * dx };
for (int i = 0; i < 8; ++i) {
    v_pos[i] = v_pos[0] + dx * glm::vec3 { (i & 1), ((i >> 1) & 1), (i >> 2) };
}
int v_hash = 0;
for (int i = 7; i >= 0 ; --i) {
    v_hash += (sdf(v_pos[i]) > 0) ? 1 : 0;
    v_hash *= 2;
}
v_hash /= 2;
int e_hash = c_EdgeStateTable[v_hash];
std::vector<int> idx_e(12, -1);
for (int j = 0; j < 12; ++j) {
    if (e_hash % 2 == 1) {
        glm::vec3 v_start, v_end, v_new;
        int       d[3]           = { 0, 0, 0 };
        d[((j >> 2) + 1) % 3]    = j & 1;
        d[((j >> 2) + 2) % 3]    = (j >> 1) & 1;
        if (grid[nx + d[0]][ny + d[1]][nz + d[2]][j >> 2] == -1) {
            v_start = v_pos[0] + dx * glm::vec3 { (float) d[0], (float) d[1], (float) d[2] };
            v_end = v_start + dx * unit[j >> 2];
            v_new = (sdf(v_end) * v_start - sdf(v_start) * v_end) / (sdf(v_end) - sdf(v_start));
```

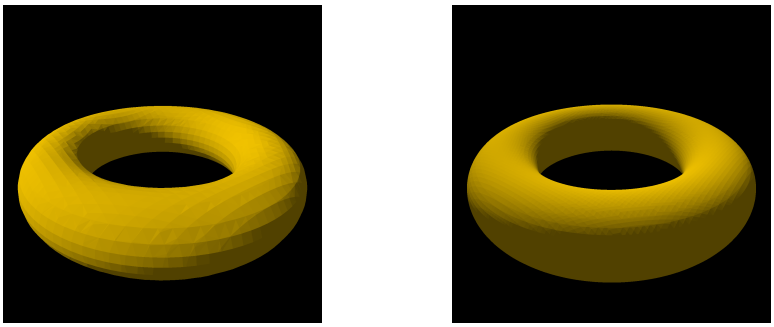
```
            grid[nx + d[0]][ny + d[1]][nz + d[2]][j >> 2] = output.Positions.size();
            output.Positions.push_back(v_new);
        }
        idx_e[j] = grid[nx + d[0]][ny + d[1]][nz + d[2]][j >> 2];
    }
    e_hash /= 2;
}
for (int k = 0; k < 5; ++k) {
    int e0 = c_EdgeOrdsTable[v_hash][3 * k], e1 = c_EdgeOrdsTable[v_hash][3 * k + 1], e2 =
    c_EdgeOrdsTable[v_hash][3 * k + 2];
    if (e0 == -1) break;
    else {
        output.Indices.push_back(idx_e[e2]);
        output.Indices.push_back(idx_e[e1]);
        output.Indices.push_back(idx_e[e0]);
    }
}
```

实现效果 对球体和环分别进行重建的效果如下:



(a) 网格尺寸 $n = 10$ (b) 网格尺寸 $n = 40$ (c) 网格尺寸 $n = 100$

图 11: 对球体进行 Marching Cubes 重建的结果



(a) 网格尺寸 $n = 40$　　(b) 网格尺寸 $n = 100$

图 12: 对环面进行 Marching Cubes 重建的结果