可视计算与交互概论-Lab2

xTryer

2024/11/10

1 Task 1:Loop Mesh Subdivision

实现思路：每一次迭代会更新的顶点由两部分组成，一部分为边生成新的顶点，另一部分为原顶点的更新。

对于边生成新的顶点，需要判断边是否是边界上的边 (判断是否存在孪生半边)，对于边界上的点，直接取中点，对于非边界上的点，用边的两个端点和以这条边为公共边的三角形的另外两个顶点加权得到新顶点。

```
    for (auto e : G.Edges()) {
        // newIndices[face index][vertex index] = index of the newly generated vertex
        newIndices[G.IndexOf(e->Face())][e->EdgeLabel()] = curr_mesh.Positions.size();
        auto eTwin                                        = e->TwinEdgeOr(nullptr);
        // eTwin stores the twin halfedge.
        if (! eTwin) {
            // When there is no twin halfedge (so, e is a boundary edge):
            // your code here: generate the new vertex and add it into curr_mesh.Positions.
            int index1=e->To();
            int index2 = e->From();

            glm::vec3 pos1 = prev_mesh.Positions[index1];
            glm::vec3 pos2 = prev_mesh.Positions[index2];

            glm::vec3 new_pos = (pos1 + pos2) * (float)0.5;
            curr_mesh.Positions.push_back(new_pos);

        } else {
            // When the twin halfedge exists, we should also record:
            //     newIndices[face index][vertex index] = index of the newly generated vertex
            // Because G.Edges() will only traverse once for two halfedges,
            //     we have to record twice.
            newIndices[G.IndexOf(eTwin->Face())][e->TwinEdge()->EdgeLabel()] = curr_mesh.
    Positions.size();
            // your code here: generate the new vertex and add it into curr_mesh.Positions.
            int index0=e->To();
            int index2 = e->From();
            int index1 = e->OppositeVertex();
            int index3 = e->TwinOppositeVertex();

            glm::vec3 pos0 = prev_mesh.Positions[index0];
            glm::vec3 pos2 = prev_mesh.Positions[index2];
```

```cpp
            glm::vec3 pos1 = prev_mesh.Positions[index1];
            glm::vec3 pos3 = prev_mesh.Positions[index3];

            glm::vec3 new_pos = (pos0 + pos2) * (float)3/(float)8 + (pos1 + pos3) * (float)1/(float)8;
            curr_mesh.Positions.push_back(new_pos);

        }
    }
```

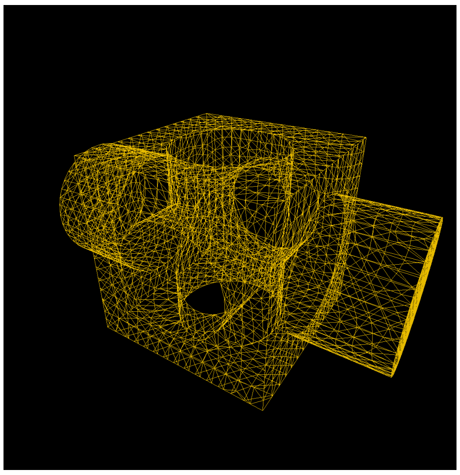
对于原顶点的更新，通过加权原顶点和与之相连的顶点得到新顶点

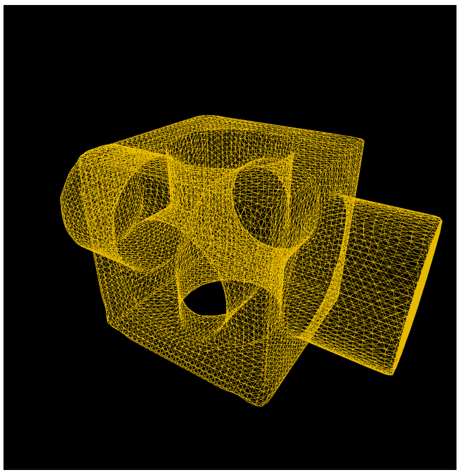
```cpp
    // Then we iteratively update currently existing vertices.
    for (std::size_t i = 0; i < prev_mesh.Positions.size(); ++i) {
        // Update the currently existing vetex v from prev_mesh.Positions.
        // Then add the updated vertex into curr_mesh.Positions.
        auto v           = G.Vertex(i);
        auto neighbors   = v->Neighbors();
        // your code here:
        int vertex_degree = neighbors.size();
        float u_   = (vertex_degree == 3) ? (float)3 / 16 : (float)3 / (8 * vertex_degree);
        glm::vec3 new_v_pos     = (1-vertex_degree*u_)*prev_mesh.Positions[i];
        for (auto neighbor_i : neighbors)
        {
            new_v_pos += prev_mesh.Positions[neighbor_i]*u_;
        }
        curr_mesh.Positions.push_back(new_v_pos);

    }
```

然后按照特定顺序连接所有顶点，产生对原先的面进一步“细分”的效果。

```cpp
    // Here we've already build all the vertices.
    // Next, it's time to reconstruct face indices.
    for (std::size_t i = 0; i < prev_mesh.Indices.size(); i += 3U) {
        // For each face F in prev_mesh, we should create 4 sub-faces.
        // v0,v1,v2 are indices of vertices in F.
        // m0,m1,m2 are generated vertices on the edges of F.
        auto v0           = prev_mesh.Indices[i + 0U];
        auto v1           = prev_mesh.Indices[i + 1U];
        auto v2           = prev_mesh.Indices[i + 2U];
        auto [m0, m1, m2] = newIndices[i / 3U];
        // Note: m0 is on the opposite edge (v1-v2) to v0.
        // Please keep the correct indices order (consistent with order v0-v1-v2)
        //     when inserting new face indices.
        // toInsert[i][j] stores the j-th vertex index of the i-th sub-face.
        std::uint32_t toInsert[4][3] = {
            // your code here:
            {m1,v0,m2},
            {m2,v1,m0},
            {m0,v2,m1},
            {m0,m1,m2}
        };
        // Do insertion.
        curr_mesh.Indices.insert(
            curr_mesh.Indices.end(),
            reinterpret_cast<std::uint32_t *>(toInsert),
            reinterpret_cast<std::uint32_t *>(toInsert) + 12U
        );
    }
```

获得结果图如下:


(a) block_original


(b) block_iteration=1

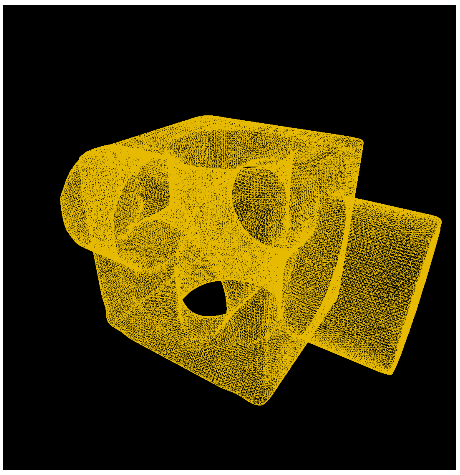
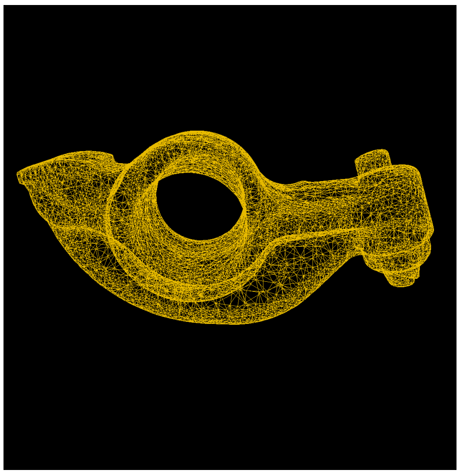

(c) block_iteration=2

图 1: Task 1:Loop Mesh Subdivision Results-Block


(a) rocker_original

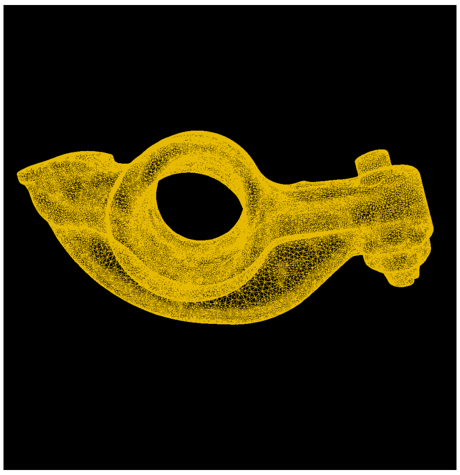

(b) rocker_iteration=1

图 2: Task 1:Loop Mesh Subdivision Results-Rocker

2 Task 2: Spring-Mass Mesh Parameterization

思路：首先需要确定三角网格的边界，按照一定顺序 (顺时针 or 逆时针)，将边界点放在一个 vector 里，

```
std::vector<int> bound_indices; //need order
int bound_cnt = 0;
int  last_bound_vertex_index=-1;
for (std::size_t i = 0; i < input.Positions.size(); ++i) {
    // get vertex proxy and properties with index i
    DCEL::VertexProxy const * v    = G.Vertex(i);
    // do something with v & pos
    if (v->OnBoundary()) {
        bound_cnt++;
        last_bound_vertex_index = i;
    }
}
int tmp_cnt =1;
bound_indices.push_back(last_bound_vertex_index);
int old_find_cur = last_bound_vertex_index;
int new_find_cur = G.Vertex(old_find_cur)->BoundaryNeighbors().first;

while (new_find_cur!=last_bound_vertex_index)
{
    bound_indices.push_back(new_find_cur);
    DCEL::VertexProxy const * v = G.Vertex(new_find_cur);
    if (v->BoundaryNeighbors().first !=old_find_cur)
    {
        old_find_cur=new_find_cur;
        new_find_cur = v->BoundaryNeighbors().first;
    }
    else
    {
        old_find_cur = new_find_cur;
        new_find_cur = v->BoundaryNeighbors().second;
    }
    tmp_cnt++;
}
```

然后初始化边界点的 uv 坐标，选择初始化为圆形边界

```
double pi    = 4 * atan(1);
float step_ = 2 * pi/bound_cnt;
for (size_t i = 0; i < bound_cnt;i++) {
    // get vertex proxy and properties with index i
    size_t                      bound_index = bound_indices[i];
    DCEL::VertexProxy const * v    = G.Vertex(bound_index);
    glm::vec3                   pos = input.Positions[bound_index];
    // do something with v & pos
    if (v->OnBoundary())
    {
        float tmp_x          = 0.5 + 0.5 * cos(step_ * i);
        float tmp_y          = 0.5 + 0.5 * sin(step_ * i);
        output.TexCoords[bound_index] = glm::vec2 { tmp_x,tmp_y };
    }
    }
```

最后通过 Gauss-Seidel 迭代求解中间点最后的 uv 坐标

```
    for (int k = 0; k < numIterations; ++k) {
        // your code here:
        for (std::size_t i = 0; i < input.Positions.size(); ++i) {
            // get vertex proxy and properties with index i
            size_t                          cnt = 0;
            glm::vec2                       tmp_tex = glm::vec2{ 0.0f, 0.0f };

            DCEL::VertexProxy const * v     = G.Vertex(i);
            // do something with v & pos
            if (!v->OnBoundary()) {
                for (auto n_index : v->Neighbors())
                {
                    //std::cout << cnt << std::endl;
                    cnt++;
                    glm::vec2 tex_ = output.TexCoords[n_index];
                    tmp_tex += tex_;
                }
                if (cnt != 0) {
                    glm::vec2 cnt_vec = glm::vec2 { cnt, cnt };
                    tmp_tex            = tmp_tex / cnt_vec;
                }
                output.TexCoords[i] = tmp_tex;
            }

        }
    }
```

获得结果图如下:

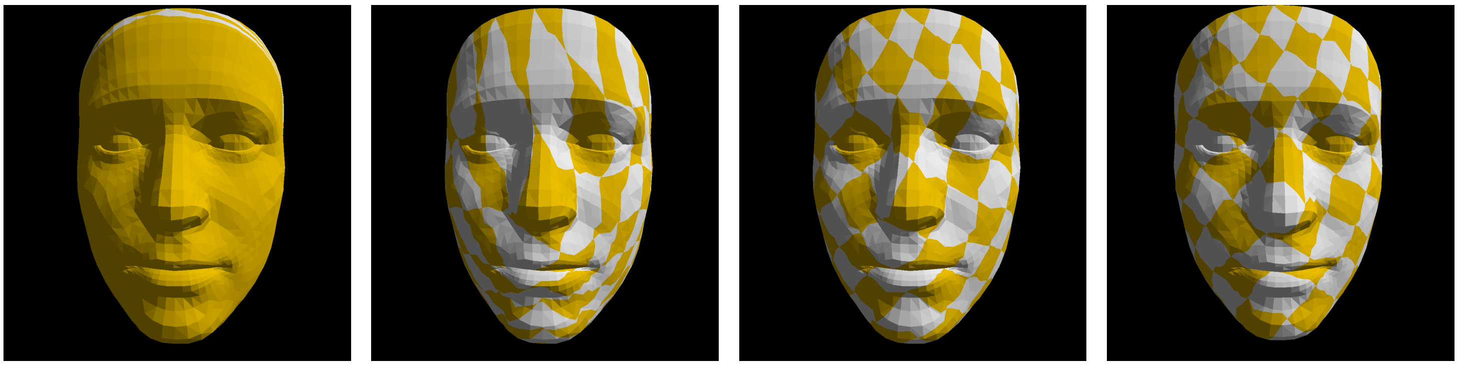

(a) iteration=0 (b) iteration=200 (c) iteration=500 (d) iteration=1000

图 3: Task 2: Spring-Mass Mesh Parameterization Results

3 Task 3: Mesh Simplification

思路：对每一个点 p_i 定义一个 4*4 的对称误差矩阵 Q_i，然后对每一条有效边 p_ip_j 进行塌缩 (要求保持网格拓扑结构)，然后利用最小化 $cost = v^T(Q_i + Q_j)v$ 求解 v 的最佳位置，然后把所有的 v 按照其 cost 进行排序，cost 小的边优先塌缩，每次塌缩后都需要更新其周围顶点的误差矩阵。

定义误差矩阵 Q：

```
    auto UpdateQ {
        [&G, &output] (DCEL::Triangle const * f) -> glm::mat4 {
            glm::mat4 Kp=glm::mat4(0);
            // your code here:
            int point_index_0 = f->VertexIndex(0);
            int point_index_1 = f->VertexIndex(1);
            int point_index_2 = f->VertexIndex(2);

            glm::vec3 triangle_edge1 = output.Positions[point_index_1] - output.Positions[
    point_index_0];
            glm::vec3 triangle_edge2 = output.Positions[point_index_2] - output.Positions[
    point_index_1];
            glm::vec3 plane_norm      = glm::normalize(glm::cross(triangle_edge1,
    triangle_edge2));
            float tmp_d                  = -glm::dot(plane_norm, output.Positions[point_index_0
    ]);
            glm::mat4 p_              = {
                glm::vec4 {plane_norm, tmp_d},
                glm::vec4 { 0, 0, 0, 0 },
                glm::vec4 { 0, 0, 0, 0 },
                glm::vec4 {0,0,0,0}
            };
            Kp = p_ * glm::transpose(p_);


            return Kp;
        }
    };
```

对每一条有效边进行塌缩，求塌缩后 v 的最佳位置和 cost 值

```
    ContractionPair ret_constraction_pair;
    glm::mat4       Q_diff = {
        glm::vec4 { Q[0] },
        glm::vec4 { Q[1] },
        glm::vec4 { Q[2] },
        glm::vec4 { 0, 0, 0, 1 }
    };
    Q_diff      = glm::transpose(Q_diff);// tanspose => diff matrix
    float uninverse_threshold = 0.001;
    glm::vec4 p_;
    float cost_;
    if (glm::determinant(Q_diff) < uninverse_threshold)
    {
        p_ = glm::vec4({ p1 + p2 ,2}) / glm::vec4 { 2.0, 2.0, 2.0,2.0 };
    }
    else
    {
        glm::mat4 tmp_inverse_matrix = glm::inverse(Q_diff);
        p_         = { tmp_inverse_matrix[3][0],
                        tmp_inverse_matrix[3][1],
                      tmp_inverse_matrix[3][2],
                       1.0 };
    }
    cost_ = glm::dot(p_, Q * p_);
    ret_constraction_pair.edge = edge;
    ret_constraction_pair.cost = cost_;
    ret_constraction_pair.targetPosition=p_;
    return ret_constraction_pair;
```

每条有效边的 cost 值进行排序，然后取最小的 cost 对应的有效边进行塌缩，塌缩后把其中一个点 v2 去除，改变 v1 的位置，然后重新计算 v1 的代价矩阵 Q_{v_i}，并更新与之相连的点的代价矩阵。

```
for (auto e : ring) {
    glm::mat4 new_kp_e_face = UpdateQ(e->Face());
    for (int tmp_index = 0; tmp_index <= 2; tmp_index++)
    {
        int target_index = e->Face()->VertexIndex(tmp_index);
        if (target_index == v1) continue;
        Qv[target_index] += (new_kp_e_face-Kf[G.IndexOf(e->Face())]);
    }

    Qv[v1] += new_kp_e_face;

    Kf[G.IndexOf(e->Face())] =new_kp_e_face;

}
```

最后由于点的代价矩阵更新，我们需要对每一条有效边重新计算 cost，然后重复以上步骤, 最终实现网格的简化。

```
    for (std::size_t i = 0; i < pairs.size(); ++i) {
        if (pairs[i].edge && (pairs[i].edge->From() == v1))
        {
            pairs[i] = MakePair(pairs[i].edge, output.Positions[v1], output.Positions[pairs[i
].edge->To()], Qv[v1] + Qv[pairs[i].edge->To()]);
        }
        else if (pairs[i].edge && (pairs[i].edge->To() == v1))
        {
            pairs[i] = MakePair(pairs[i].edge, output.Positions[pairs[i].edge->From()], output
.Positions[v1], Qv[v1] + Qv[pairs[i].edge->From()]);
        }
    }
```

获得结果图如下:

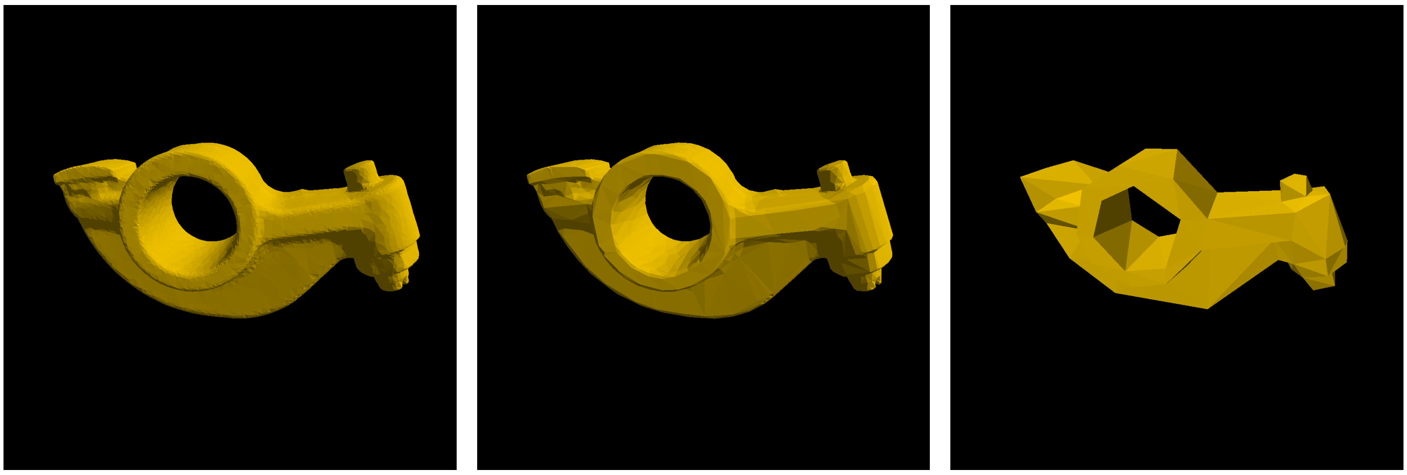

(a) Rocker Simplify=0 (b) Rocker Simplify=2 (c) Rocker Simplify=4

图 4: Task 3: Mesh Simplification Results-Rocker

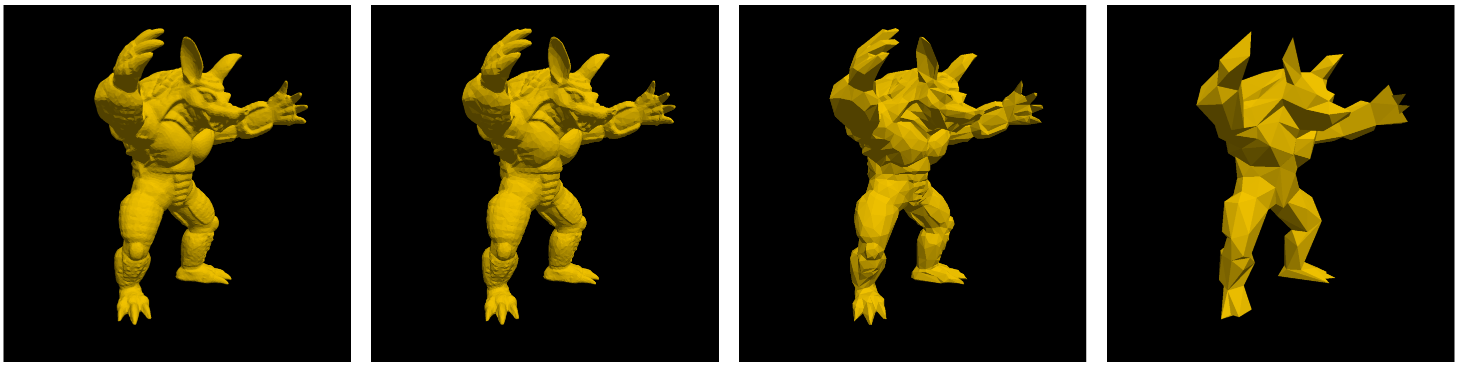

(a) Arma Simplify=0 (b) Arma Simplify=1 (c) Arma Simplify=3 (d) Arma Simplify=4

图 5: Task 3: Mesh Simplification Results-Arma

4 Task 4: Mesh Smoothing

思路：对每个顶点，计算相邻顶点的加权平均，其中根据加权方式可以分为两种网格平滑:

$$v_i' = \frac{\Sigma_{j \in N(i)} w_{ij} v_j}{\Sigma_{j \in N(i)} w_{ij}}$$

- Uniform Laplacian: 相邻顶点权重均为 1
- Cotangent Laplacian: 相邻顶点权重使用余切权重 $w_{ij} = cot\alpha_{ij} cot\beta_{ij}$

然后对每个顶点进行更新：

$$v_i = (1 - \lambda) * v_i + \lambda * v_i$$

首先计算一个角的 cotangent 值:

```
    static constexpr auto GetCotangent {
        [] (glm::vec3 vAngle, glm::vec3 v1, glm::vec3 v2) -> float {
            // your code here:
            glm::vec3 edge1=v1-vAngle;
            glm::vec3 edge2=v2-vAngle;
            float     angle_cos = (glm::dot(glm::normalize(edge1),glm::normalize( edge2)));
            if (angle_cos <= 0) return 0.0f; //<0 ret 0
            else if (angle_cos == 1) return 10000.0f;
            float angle_cotangent = angle_cos / sqrt(1 - angle_cos * angle_cos);

            return angle_cotangent;
        }
    };
```

根据 useUniformWeight? 来决定权重的计算方式，然后对每个点计算出新顶点的位置。

```
    Engine::SurfaceMesh curr_mesh = prev_mesh;
    for (std::size_t i = 0; i < input.Positions.size(); ++i) {
        // your code here: curr_mesh.Positions[i] = ...
        glm::vec3 new_pos = { 0, 0, 0 };
        DCEL::VertexProxy const * v        = G.Vertex(i);
        glm::vec3  old_pos      = curr_mesh.Positions[i];
        // do something with v & pos
        float w_sum = 0;
        for (auto edge_ : v->Ring()) {
            auto      twin_edge           = edge_->TwinEdge();
            auto pointj_index = edge_->To();
            auto      point1_index        = edge_->NextEdge()->To();
            auto point2_index = twin_edge->NextEdge()->To();
            glm::vec3 pj_pos        = curr_mesh.Positions[pointj_index];
            glm::vec3 p1_pos        = curr_mesh.Positions[point1_index];
            glm::vec3 p2_pos        = curr_mesh.Positions[point2_index];
            float     get_cot_alpha       = GetCotangent(p1_pos, pj_pos, old_pos);
            float     get_cot_beta     = GetCotangent(p2_pos, pj_pos, old_pos);
            float     w_ij                = (useUniformWeight=true)?1:get_cot_alpha +
   get_cot_beta;
            new_pos += w_ij * pj_pos;
            w_sum += w_ij;
        }
        new_pos = new_pos / w_sum;
        new_pos = (1 - lambda) * old_pos + lambda * new_pos;
        curr_mesh.Positions[i] = new_pos;
    }
```

再进行反复的迭代，得到最终平滑处理后的三维网格图像。

获得结果图如下:

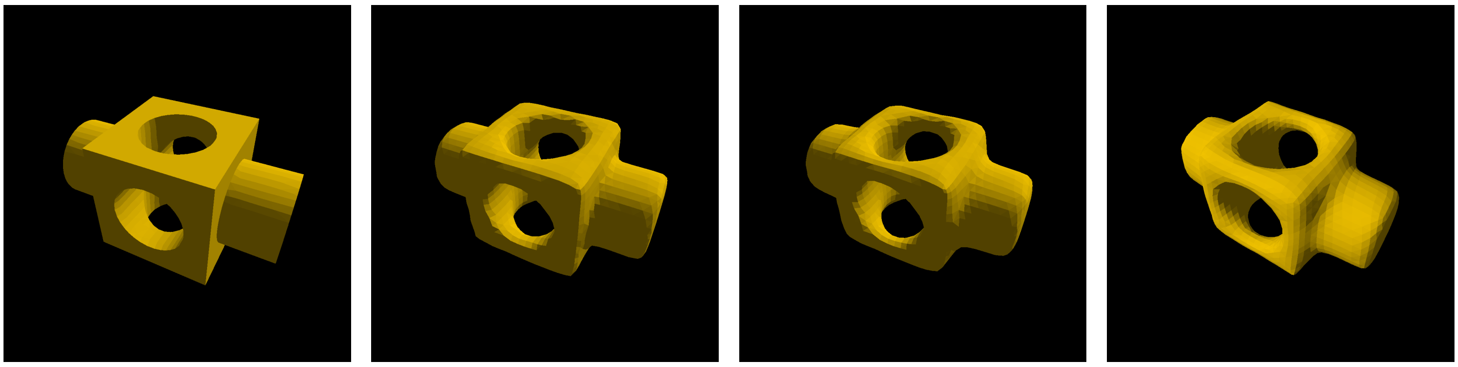

(a) Sphere original (b) Sphere iteration=4 (c) Sphere iteration=8 (d) Sphere iteration=10

图 6: Task 4: Mesh Smoothing Results-Sphere

5 Task 5: Marching Cubes

思路：遍历范围内的每一个小立方体，对其 8 个顶点分别使用 sdf 函数求其到等值面的距离，进而判断哪些点在对象内部，哪些点在对象外部。(使用 v 变量对点的分布情况进行存储)

```
    glm::vec3 point_pos = unit_step[0] * float(x_)*dx + unit_step[1] * float(y_)*dx +
   unit_step[2] * float(z_)*dx + grid_min;
    glm::vec3 points_positions[8];
    uint32_t  v = 0;
    for (int i = 0; i < 8; i++)
    {
        points_positions[i] = point_pos+glm::vec3 { (i & 1) * dx,((i >> 1) & 1) * dx,(i >> 2)
   * dx };
    }
    for (int i = 0; i < 8; i++)
    {
        //printf("%d %f\n", i, sdf(points_positions[i]));
        if (sdf(points_positions[i]) > 0.0)
        {
            v += (1 << i);
        }
    }
```

根据点的分布情况，进而根据 EdgeStateTable 判断其 12 条边上是否存在 mesh 顶点 (存在情况存在 edge_states 这个变量中)，对于存在 mesh 顶点的边，我们利用该边的两个顶点及这两个顶点的 sdf 值来进行线性插值，得到对应的 mesh 顶点。

```
glm::vec3 interpolation_get_p_(const glm::vec3  p1, const glm::vec3  p2, float p1_sdf_val,
   float p2_sdf_val) {
    float sdf_val_21 = p2_sdf_val - p1_sdf_val;
    if (std::abs(p1_sdf_val) < 0.00001) return p1;
    if (std::abs(p2_sdf_val) < 0.00001) return p2;
    if (std::abs(sdf_val_21) < 0.00001) return p1;

    glm::vec3 tar_pos = (p2_sdf_val * p1 - p1_sdf_val * p2) / sdf_val_21;

    return tar_pos;
}

if (((edge_states&(1<<j))==0))continue; // no mesh point
glm::vec3 edge_from = points_positions[0] + dx * (j & 1) * unit_step[((j >> 2) + 1) % 3] + dx
    * ((j >> 1) & 1) * unit_step[((j >> 2) + 2) % 3];
glm::vec3 edge_to   = edge_from + unit_step[j >> 2] * dx;
float      p1_sdf_val = sdf(edge_from);
float      p2_sdf_val = sdf(edge_to);

glm::vec3 edge_tar_pos =interpolation_get_p_(edge_from,edge_to,p1_sdf_val,p2_sdf_val);
```

由于在我们移动 cube 的过程中，会存在我们重复计算了某些 edge 的情况，因此我们需要考虑到这种情况，避免重复将同一条 edge 计算出的 mesh_point 重复加入到 output 中

对此，我们在一开始创建了一个哈希表，具体是把一个立方体的 cube_id 映射到其对应的上的 mesh_point 组成的 vector, 这个 vector 中存储了 mesh_point 和它最后在 ouput 中的顶点编号，这样我们就避免了重复导入 position 的问题。

哈希表：

```
    std::unordered_map < int, std::vector<std::pair<int,glm::vec3>>> cube_mesh_points_map;
    //cube_id = x*n*n+y*n+z to index & tar_point
    int        point_cnt=0;//record mesh point num
```

检查临近的 cube 以查询该有效边是否已经被计算过：

```
bool close_union_check(glm::vec3 point1, glm::vec3 point2,float threshold)
{
    glm::vec3 point_distance = point1-point2;

    return (std::abs(point_distance.x) < threshold) && (std::abs(point_distance.y) < threshold
    ) && (std::abs(point_distance.z) < threshold);
}//check whether the points pair is close enough to think them are the same
    if (x_ > 0)
    {
        int nearby_cube = (x_ - 1) * n * n + y_ * n + z_;
        for (auto nearby_edge_pair : cube_mesh_points_map[nearby_cube])
        {
            if (close_union_check(edge_tar_pos, nearby_edge_pair.second, dx/500))
            {
                edge_tar_pos = nearby_edge_pair.second;
                position_index = nearby_edge_pair.first;
                break;
            }
        }
    }
    if (y_ > 0 && position_index == -1)
    {
        int nearby_cube = (x_ ) * n * n + (y_-1) * n + z_;
        for (auto nearby_edge_pair : cube_mesh_points_map[nearby_cube]) {
            if (close_union_check(edge_tar_pos, nearby_edge_pair.second, dx/500)) {
                edge_tar_pos    = nearby_edge_pair.second;
                position_index = nearby_edge_pair.first;
                break;
            }
        }
    }
    if (z_ > 0 && position_index == -1)
    {
        int nearby_cube = (x_) * n * n + y_ * n + z_-1;
        for (auto nearby_edge_pair : cube_mesh_points_map[nearby_cube]) {
            if (close_union_check(edge_tar_pos, nearby_edge_pair.second, dx/500)) {
                edge_tar_pos    = nearby_edge_pair.second;
                position_index = nearby_edge_pair.first;
                break;
            }
        }
    }
    edge_point_map[j] = std::make_pair(position_index, edge_tar_pos);
```

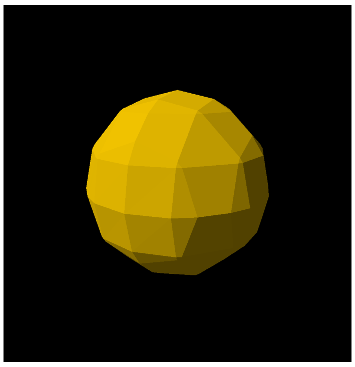
最后利用 EdgeOrdsTable 来把产生的新的 mesh_point 加入到 output.Indices 和 output.Positions 里，如果这个点已经出现过，那么只需要加入到 Indices 里，否则还需要作为新点加入到 Positions 里。

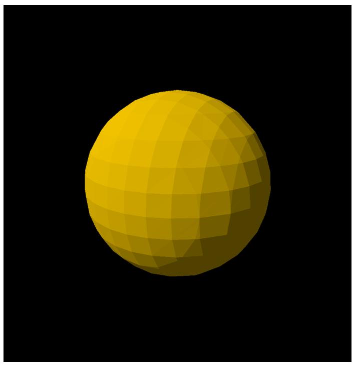
```
    int k = 0;
while (c_EdgeOrdsTable[v][3 * k] != -1)
{
    int index[3] = { c_EdgeOrdsTable[v][3 * k],
                     c_EdgeOrdsTable[v][3 * k+1],
                     c_EdgeOrdsTable[v][3 * k+2] };

    for (int j = 2; j >=0; j--)
    {
        if (edge_point_map[index[j]].first != -1)//already in
        {
            cube_mesh_points_map[x_ * n * n + y_ * n + z_].push_back(std::make_pair(edge_point_map[index[j]].first, edge_point_map[index[j]].second));
            output.Indices.push_back(edge_point_map[index[j]].first);
        }
        else
        {
            cube_mesh_points_map[x_ * n * n + y_ * n + z_].push_back(std::make_pair(point_cnt, edge_point_map[index[j]].second));
            output.Indices.push_back(point_cnt);
            output.Positions.push_back(edge_point_map[index[j]].second);
            glm::vec3 tmp_pos         = edge_point_map[index[j]].second;
            edge_point_map[index[j]] = std::make_pair(point_cnt,tmp_pos);
            point_cnt++;
        }

    }
```

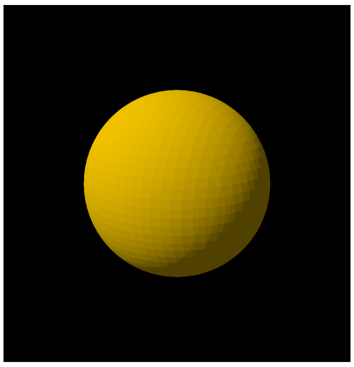
```
        k++;
}
```

最后不断迭代，实现把隐式表面转为显式的 Mesh 表示。

获得结果图如下:


(a) Sphere resolution=10


(b) Sphere resolution=20


(c) Sphere resolution=50

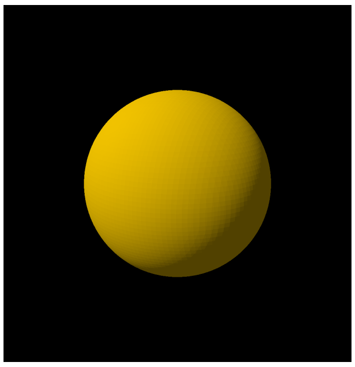
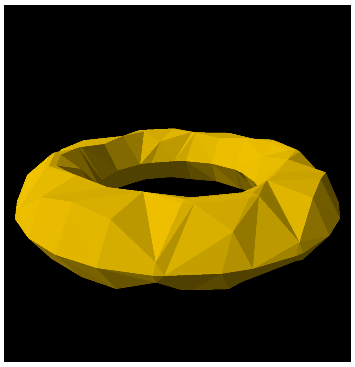
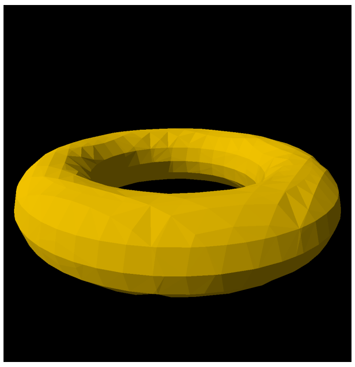
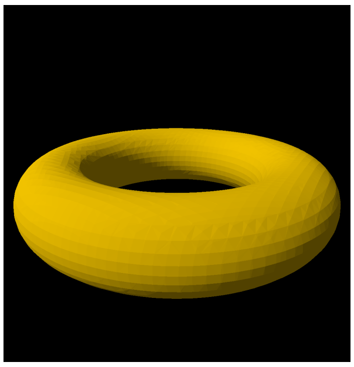

(d) Sphere resolution=100

图 7: Task 5: Marching Cubes Results-Sphere


(a) Torus resolution=10


(b) Torus resolution=20


(c) Torus resolution=50

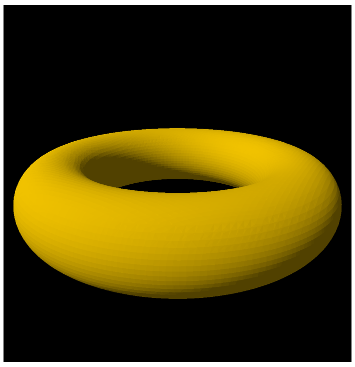

(d) Torus resolution=100

图 8: Task 5: Marching Cubes Results-Torus