Lab2 Report

2300012929 尹锦润

1. SubdivisionMesh

对于每次迭代，我们首先根据公式计算已经有的节点新坐标：

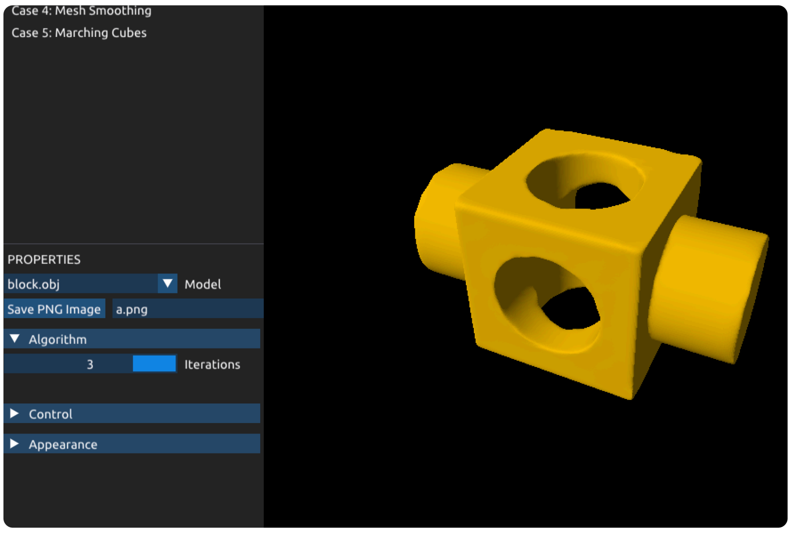
```cpp
for (std::size_t i = 0; i < prev_mesh.Positions.size(); ++i) {
    // Update the currently existing vetex v from prev_mesh.Positions.
    // Then add the updated vertex into curr_mesh.Positions.
    auto v           = G.Vertex(i);
    auto neighbors   = v->Neighbors();
    int n = neighbors.size();
    float u = n == 3 ? 3. / 16 : 3. / 8 / n;
    auto newpos = prev_mesh.Positions[i];
    newpos *= (1 - n * u);
    for (auto t : neighbors) {
        newpos += u * prev_mesh.Positions[t];
    }
    curr_mesh.Positions.push_back(newpos);
}
```

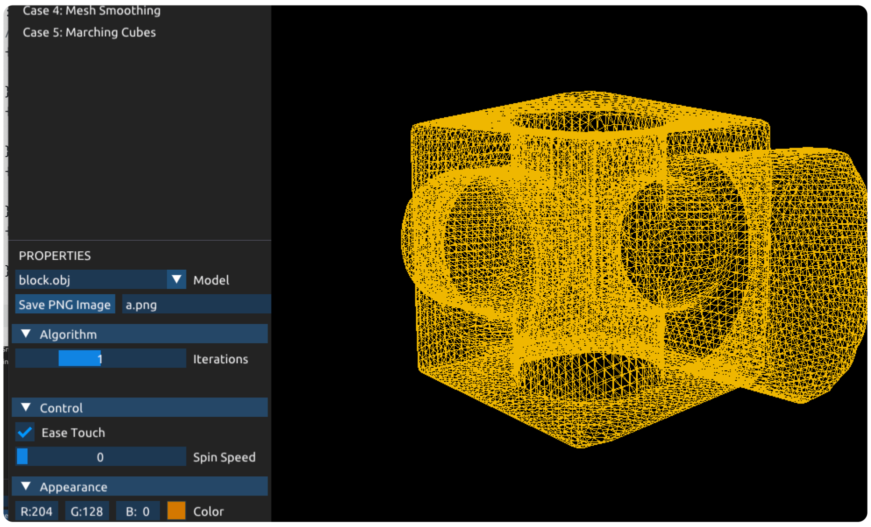
接着对于边我们计算其边点坐标：

```cpp
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):
        curr_mesh.Positions.push_back((prev_mesh.Positions[e->To()] +
prev_mesh.Positions[e->From()]) / (float)2.);
    } 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();
        glm::vec3 v0 = prev_mesh.Positions[e->To()];
        glm::vec3 v2 = prev_mesh.Positions[e->From()];
        glm::vec3 v1 = prev_mesh.Positions[e->OppositeVertex()];
        glm::vec3 v3 = prev_mesh.Positions[e->TwinEdge()->OppositeVertex()];
        glm::vec3 nv = (float) 3. / 8 * (v0 + v2) + (float) 1. / 8 * (v1 + v3);
        curr_mesh.Positions.push_back(nv);
    }
}
```

最后根据坐标进行新的连边：

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





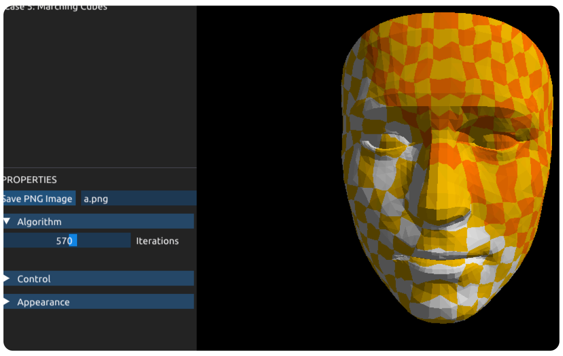
2. Spring-Mass Mesh Parameterization

首先，对于边界点设置好圆映射坐标，注意根据群里面的描述，需要保证坐标都是正数，不然会出现问题：

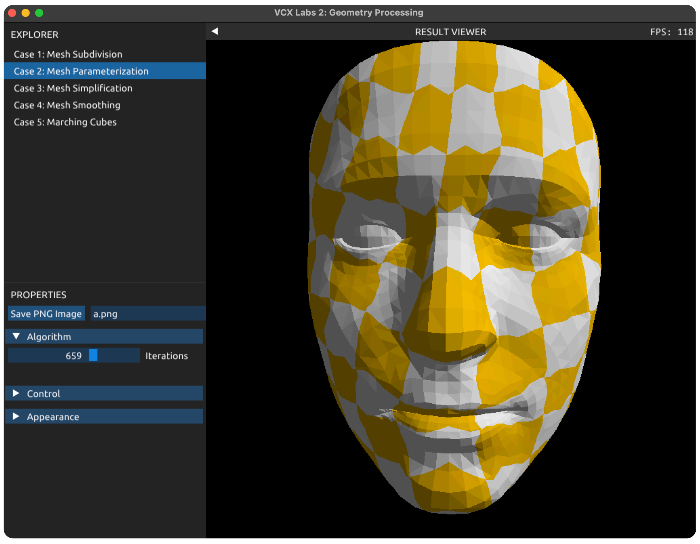
```cpp
std::vector < int > pot;
std::vector < bool > mark(input.Positions.size(), 0);
for(auto e : G.Edges()) {
    if(!e->TwinEdgeOr(nullptr)) {
        pot.push_back(e->From());
        pot.push_back(e->To());
        mark[e->From()] = mark[e->To()] = 1;
    }
}
sort(pot.begin(), pot.end(), [&](int x, int y) {
    return atan2(input.Positions[x].x, input.Positions[x].y) <
            atan2(input.Positions[y].x, input.Positions[y].y);
});
for(int i = 0; i < pot.size(); i++) {
    output.TexCoords[pot[i]] = glm::vec2{ cos(2 * M_PI * i / pot.size()) / 2 + 1, sin(2 * M_PI * i / pot.size()) / 2 + 1};
}
```

进行迭代求解：

```
for (int k = 0; k < numIterations; ++k) {
    for(int i = 0; i < input.Positions.size(); i++) if(!mark[i]) {
        glm::vec2 nx(0, 0);
        auto neigh = G.Vertex(i)->Neighbors();
        float ct = 1. / neigh.size();
        for (auto v : neigh)
            nx += output.TexCoords[v];
        nx *= ct;
        output.TexCoords[i] = nx;
    }
}
```



（如果坐标存在负数就是这样）



正常版。

3. Mesh Simplification

根据论文中的公式，计算出来 K_p 矩阵：

```cpp
auto UpdateQ {
    [&G, &output] (DCEL::Triangle const * f) -> glm::mat4 {
        glm::mat4 Kp;
        // your code here:
        glm::vec3 u = output.Positions[f->VertexIndex(0)];
        glm::vec3 v = output.Positions[f->VertexIndex(1)];
        glm::vec3 w = output.Positions[f->VertexIndex(2)];
        glm::vec3 ex = { -1., -1., -1. };
        glm::vec3 n = glm::inverse(glm::transpose(glm::mat3(u, v, w))) * ex;
        double scale = sqrt((n.x * n.x + n.y * n.y + n.z * n.z));
        n /= scale;   // a ^ 2 + b ^ 2 + c ^ 2 + d = 1;
        glm::vec4 p = { n.x, n.y, n.z, 1. / scale };
        for(int i = 0; i < 4; i++) for(int j = 0; j < 4; j++) Kp[i][j] = p[i] * p[j];
        return Kp;
    }
};
```

对于 连边的 `pair` 计算时候需要考虑 `mat4` 的初始化顺序问题 **(列优先)**：

```cpp
static constexpr auto MakePair {
    [] (DCEL::HalfEdge const * edge,
        glm::vec3 const & p1,
        glm::vec3 const & p2,
        glm::mat4 const & Q
    ) -> ContractionPair {
        // your code here:
        glm::mat4 tQ = glm::transpose(Q);
        tQ[0][3] = tQ[1][3] = tQ[2][3] = 0; tQ[3][3] = 1;
        glm::vec4 b = { 0, 0, 0, 1 };
        glm::vec4 v = {(p1 + p2) / 2.f, 1};
        if(glm::determinant(tQ) >= 0.001) v = glm::inverse(tQ) * b;
        float cost = glm::dot(v, Q * v);
        return { edge, v, cost };
    }
};
```

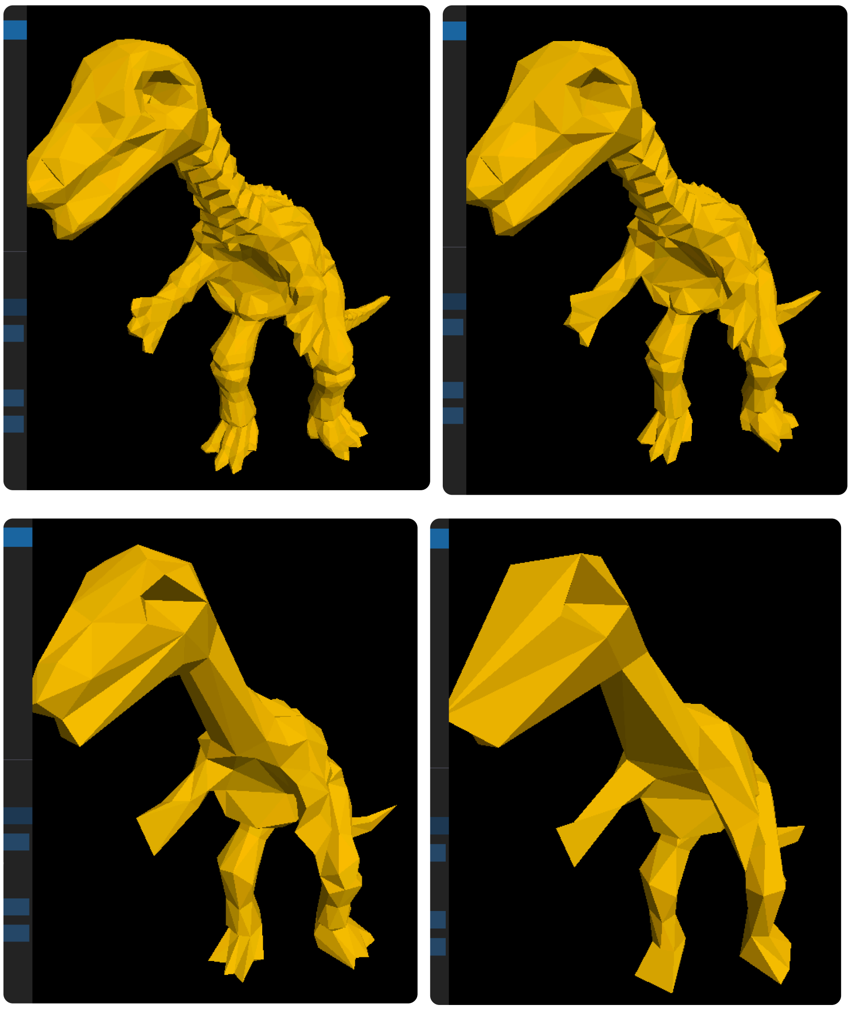
接着就是不断迭代，不断处理 `pair` 的坍塌。

坍塌之后需要考虑对于 `v1` 进行 `ring` 上权重的更新：

```cpp
for (auto e : ring) {
    auto f                  = e->Face();
    auto Q                  = UpdateQ(f);
    Qv[e->From()] += Q - Kf[G.IndexOf(f)];
    Qv[e->To()] += Q - Kf[G.IndexOf(f)];
    Qv[v1] += Q;
    Kf[G.IndexOf(f)] = Q;
}
```

最后，对于那些 Q 已经更新的点，我们继续考虑它周围的环（*e*2-tring），来集体更新 pairs。

```cpp
for (auto e1 : ring) {
    auto vv1 = e1->From();
    auto tring   = G.Vertex(vv1)->Ring();
    for (auto e2 : tring){
        if (!G.IsContractable(e2->NextEdge())){
            pairs[pair_map[G.IndexOf(e2->NextEdge())]].edge=nullptr;
        }
        else{
            auto vv2 = e2->To();
            auto tpair = MakePair(e2->NextEdge(), output.Positions[vv1], output.Positions[vv2], Qv[vv1] + Qv[vv2]);
            pairs[pair_map[G.IndexOf(e2->NextEdge())]].targetPosition = tpair.targetPosition;
            pairs[pair_map[G.IndexOf(e2->NextEdge())]].cost = tpair.cost;
        }
    }
}
```



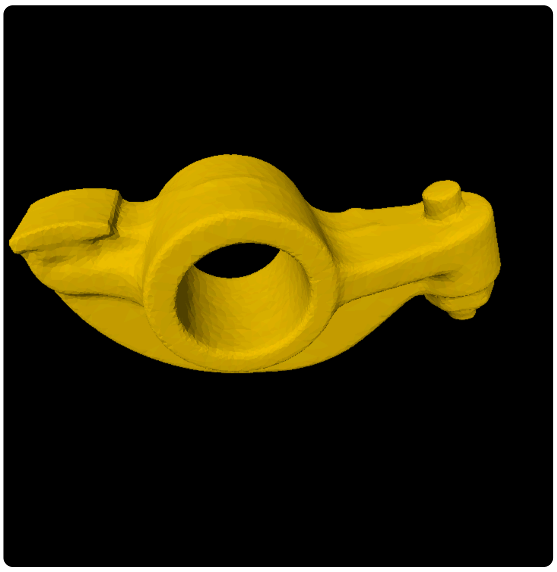
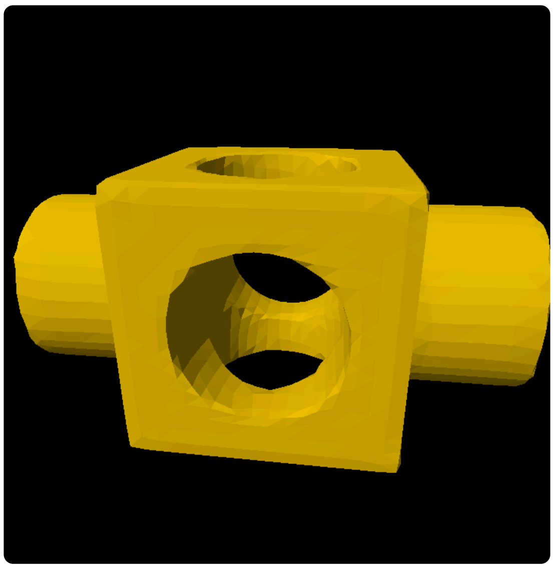
4. Mesh Smoothing

首先定义好计算 cot 的方式:

```cpp
static constexpr auto GetCotangent {
    [] (glm::vec3 vAngle, glm::vec3 v1, glm::vec3 v2) -> float {
        // your code here:
        glm::vec3 a = v1 - vAngle;
        glm::vec3 b = v2 - vAngle;
        float l1 = sqrt(glm::dot(a, a)), l2 = sqrt(glm::dot(b, b));
        float vcos = glm::dot(a, b) / l1, vsin = sqrt(1 - vcos * vcos);
        return fabs(vcos / vsin);
    }
};
```

在每次迭代每个点的过程中我们可以利用边的反边、顶点关系来计算需要的角度:

```
for(auto e : edges) {
    auto t = e->PrevEdge();
    int x = t->To(), y = e->To(), z = t->TwinEdge()->NextEdge()->To();
    if(useUniformWeight) {
        tot += 1;
        newpos += input.Positions[x];
    }
    else {
        float ret = GetCotangent(input.Positions[i], input.Positions[x],
input.Positions[y]) +
            GetCotangent(input.Positions[i], input.Positions[x], input.Positions[z]);
        newpos += ret * input.Positions[x];
        tot += ret;
    }
}
curr_mesh.Positions[i] = lambda * (newpos / tot) + input.Positions[i] * (1 - lambda);
```

(均为 5 iteration, 0.4 smoothness)

5. Marching Cubes

根据 readfile 将函数输入进去:

```
auto getNodePos = [&](glm::vec3 t, int i) {
    t.x += (i & 1) * dx;
    t.y += (i >> 1 & 1) * dx;
    t.z += (i >> 2 & 1) * dx;
    return t;
};
auto unit = [&](int i) {
    if(i == 0) return glm::vec3(1, 0, 0);
    if(i == 1) return glm::vec3(0, 1, 0);
    if(i == 2) return glm::vec3(0, 0, 1);
    return glm::vec3(0, 0, 0);
};
auto getEdgePos = [&](glm::vec3 t, int j) {
    return t + dx * (j & 1) * unit(((j >> 2) + 1) % 3) + dx * ((j >> 1) & 1) * unit(((j >> 2) + 2) % 3);
};
```

然后获取每次的状态，根据位数所以 是 7-0:

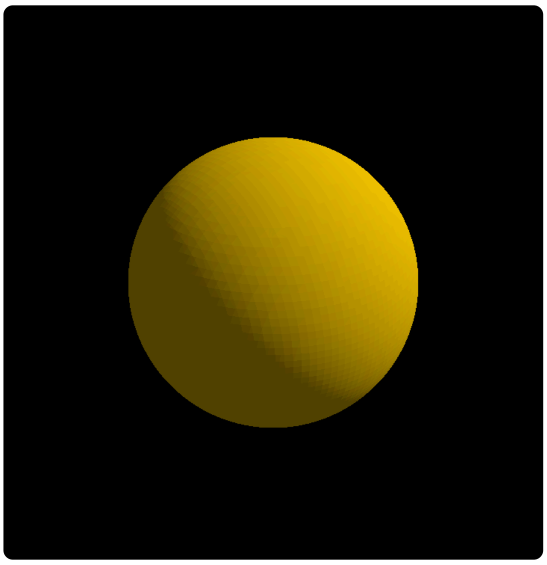
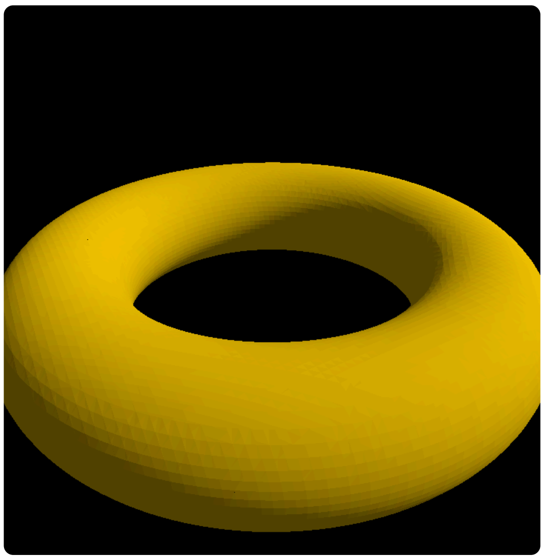
```
glm::vec3 v0 = grid_min + glm::vec3(dx * x, dx * y, dx * z);
for(int s = 7; s >= 0; s--) stu = stu << 1 | (sdf(getNodePos(v0, s)) >= 0);
```

根据边的状态我们生成需要的边点的位置（**需要注意插值的公式**）:

```
int edgestu = c_EdgeStateTable[stu];
for(int s = 0; s < 12; s++) if(edgestu >> s & 1) {
    pos[s] = output.Positions.size();
    glm::vec3   stpos = getEdgePos(v0, s),
                edpos = stpos + dx * unit(s >> 2);
    float v1 = sdf(edpos), v2 = sdf(stpos);
    glm::vec3   edgepos = ((v2) * edpos + (-v1) * stpos) / (v2 - v1);
    output.Positions.push_back(edgepos);
    glm::vec3   stn = getNormal(stpos),
                edn = getNormal(edpos),
                normal = ((v2) * edn + (-v1) * stn) / (v2 - v1) ;
    output.Normals.push_back(normal);
}
```

最后按照预设的规则进行连边:

```
for(int t = 0; t < 6; t++) if(c_EdgeOrdsTable[stu][t * 3] != -1) {
    int a = c_EdgeOrdsTable[stu][t * 3],
        b = c_EdgeOrdsTable[stu][t * 3 + 1],
        c = c_EdgeOrdsTable[stu][t * 3 + 2];
    output.Indices.push_back(pos[c]);
    output.Indices.push_back(pos[b]);
    output.Indices.push_back(pos[a]);
}
```

均为分辨率拉满的状态。