Lab02

Task 1

这个Task也即让我们实现课件中的Loop Subdivision，更新点的时候根据规则，旧的点的新坐标为 $v' = \begin{cases} \frac{3}{4}v + \frac{1}{4}\sum_{v_i \in boundary \cap neighbor} v_i & v \in boundary \\ (1-n\beta)v + \beta\sum_{v_i \in neighbor} v_i, & v \notin boundary \end{cases}'$ 其中n为这个顶点的度数，$\beta = \begin{cases} \frac{3}{16} & n = 3 \\ \frac{3}{8n} & other \end{cases}$。计算的时候只要把这个公式翻译成代码即可，最后把计算出的`update_v``push_back()`进入`curr_mesh.Positions`即可。因此这一部分代码实现为

```
glm::vec3 updated_v = glm::vec3(0.0f);
if(v->OnBoundary())
{
    glm::vec3 boundary_neighbor_sum(0.0f);
    int boundary_count = 0;
    for(auto neighbor_idx:neighbors)
    {
        auto neighbor_vertex = G.Vertex(neighbor_idx);
        if(neighbor_vertex->OnBoundary())
        {
            boundary_neighbor_sum += prev_mesh.Positions[neighbor_idx];
            boundary_count++;
        }
    }
    updated_v=0.75f*prev_mesh.Positions[i]+0.25f*
(boundary_neighbor_sum/static_cast<float>(boundary_count));
    }
else
{
    float n = static_cast<float>(neighbors.size());
    float beta = (n == 3.0f) ? 3.0f / 16.0f : 3.0f / (8.0f * n);
    glm::vec3 neighbor_sum(0.0f);
    for(auto neighbor_idx:neighbors)
        neighbor_sum += prev_mesh.Positions[neighbor_idx];
    updated_v = (1.0f - n * beta) * prev_mesh.Positions[i] + beta * neighbor_sum;
}
curr_mesh.Positions.push_back(updated_v);
```

而当我们遍历每一条边的时候，新的点的公式则为$v' = \begin{cases} \frac{1}{2}(v_1+v_2) & v \in boundary \\ \frac{3}{8}(v_1+v_2)+\frac{1}{8}(v_3+v_4) & v \notin bounadary \end{cases}$，其中$v_1$，$v_2$为这条边的原顶点，$v_3$，$v_4$（$e$不为边界边情况）则是这条边**opposite**的两个顶点（如课件图中所示）。那么如何判定一条边是不是边界呢？答案就是它没有`e->TwinEdge()`。于是我们的代码就应该是

```
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.
    glm::vec3 new_vertex = 0.5f * (prev_mesh.Positions[e->From()] +
prev_mesh.Positions[e->To()]);
    curr_mesh.Positions.push_back(new_vertex);
}
```

```
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.
    glm::vec3 p1=prev_mesh.Positions[e->From()];
    glm::vec3 p2=prev_mesh.Positions[e->To()];
    glm::vec3 p3=prev_mesh.Positions[e->NextEdge()->To()];
    glm::vec3 p4=prev_mesh.Positions[eTwin->NextEdge()->To()];
    glm::vec3 new_vertex = 0.375f * (p1 + p2) + 0.125f * (p3 + p4);
    curr_mesh.Positions.push_back(new_vertex);
}
```

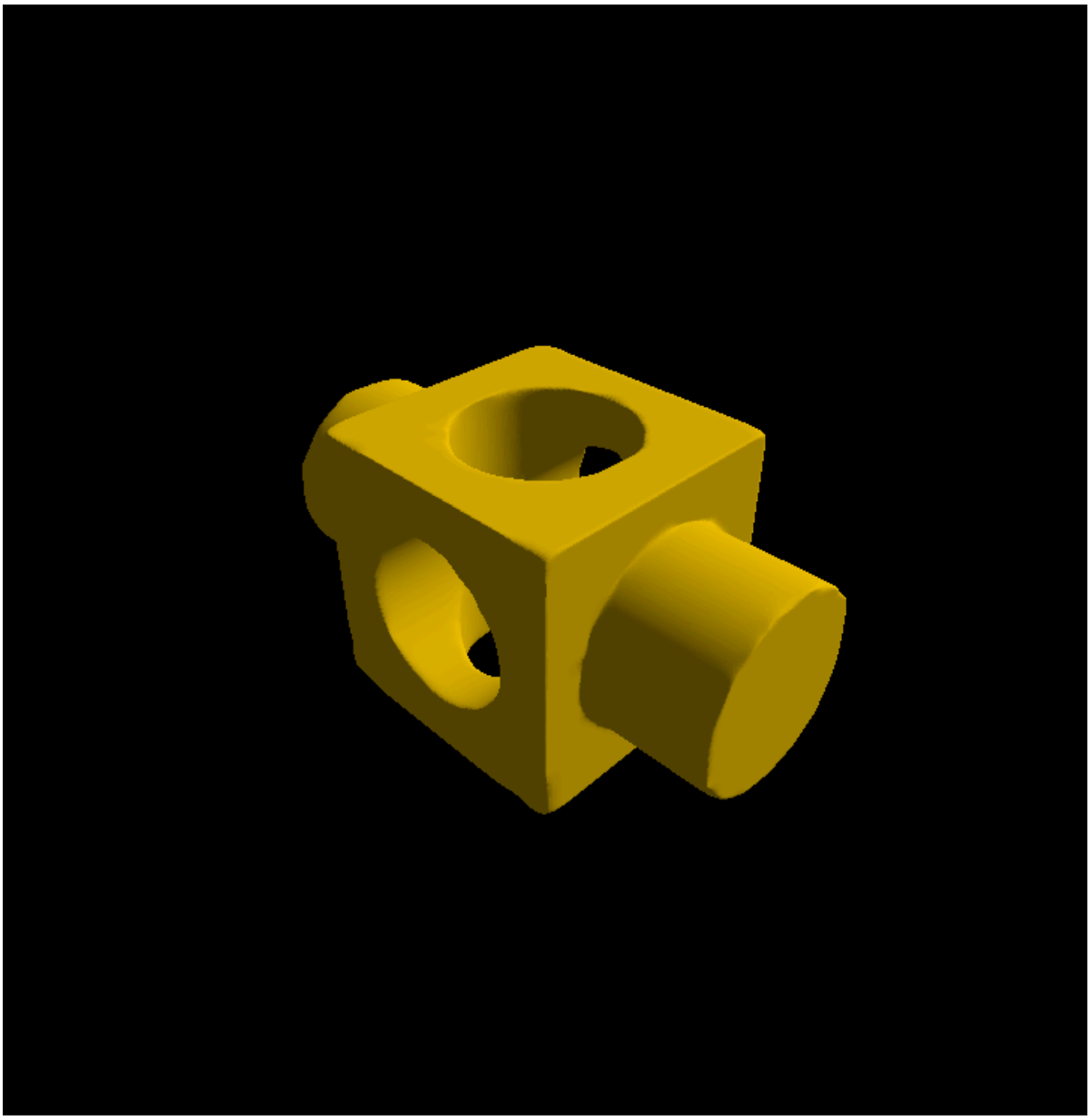
在构建完我们的顶点后，剩余的就是把顶点连起来形成新的面了。根据提示，我们的`toInsert`矩阵的每一行是一个新的面，顶点顺序要和原来的`v0,v1,v2`保持一致。我们通过上面的方式生成的点的相对关系应该如图所示：

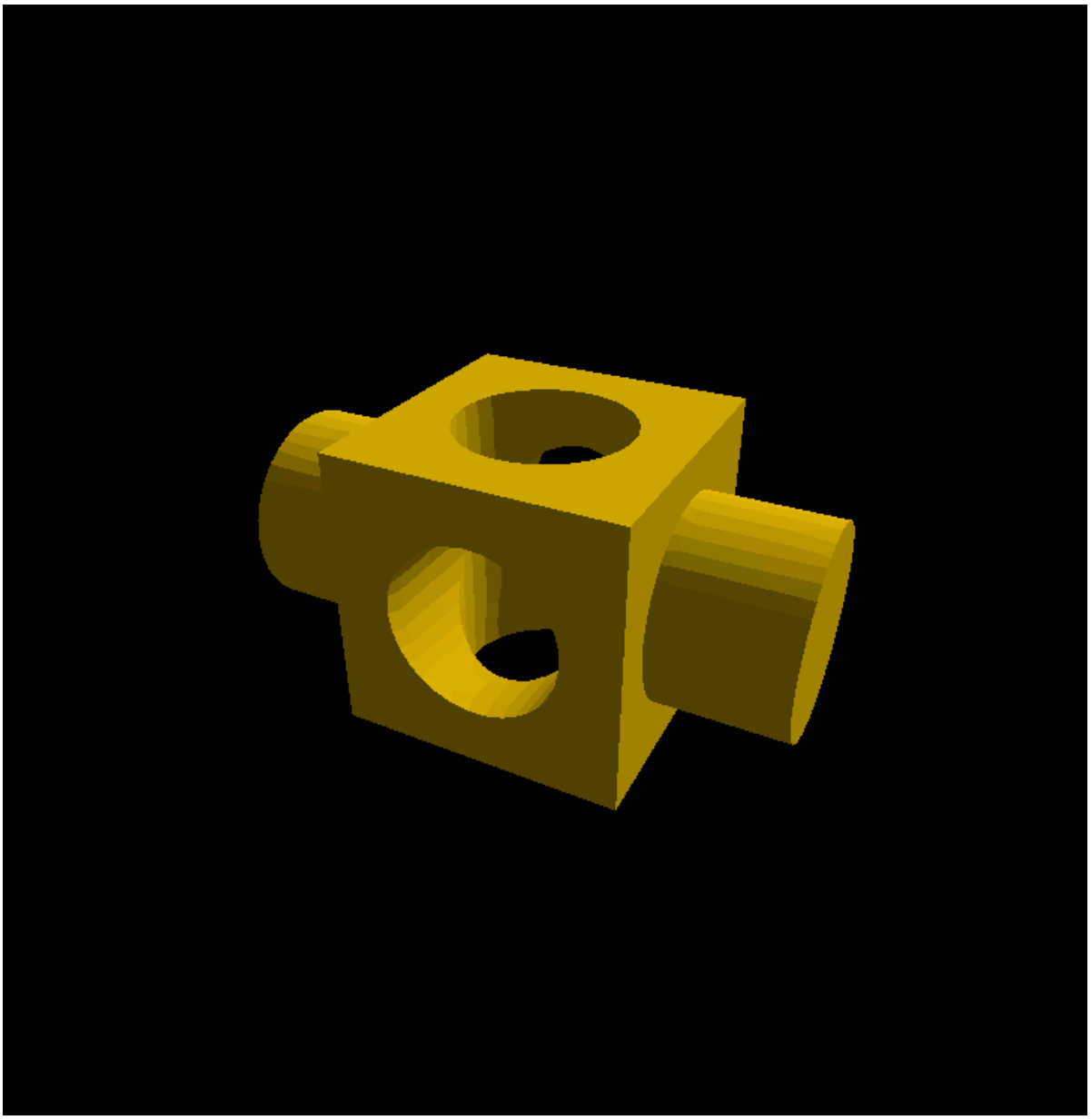
```
      v0
     / \
    /   \
   m2---m1
  / \   / \
 /   \ /   \
v1----m0----v2
```

如果我们要保持顺序一致，那么在这张图里就应该是逆时针方向，因此

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

这样就完成了面的构建，也同时完成了一整个更新的过程。此处仅展示block.obj的iteration设为0和3的区别，剩余图像请见images文件夹。





可以看到，在经过了3轮次的迭代过后，物体的表面光滑了很多。

Task2

在这个task中，我选择的边界条件是把边界点展平到$[0,1]\times[0,1]$为外接正方形的圆上（即圆心为$(\frac{1}{2},\frac{1}{2})$,半径$R=\frac{1}{2}$），并且边节点之间的间距正比于它们原本三维空间的距离。为实现此效果，需要分三步走：

- 通过遍历所有的边，找到全部边界点（有序）
- 计算长度和累计长度
- 映射到圆上

对于第一步的遍历，需要先找出第一个边界边（`e->TwinEdgeOr(nullptr)==nullptr`），那么这条边的起点和终点都是边界点。随后以这条边为起点，遍历所有的边，每次先执行`curr_edge=curr_edge->NextEdge();`，这样可以保证当前边的起点**一定是边界点**，也就至少有一个边界边以这个点为起点；随后循环 `curr_edge=curr_edge->TwinEdge()->NextEdge();`这就好像是我们从一条边界边出发，每次像拨动时针一样找寻**同一个顶点**出发的另外一条边，直到找到这个边界边。且这样找到的边界边都是首尾相连的，也就达成了我们想要有序遍历所有的边界点的目标。

获得了所有边界点的索引构成的数组后，我们把相邻点之间的距离存入数组`edge_length`，最后用圆上的圆心角正比于弧长就可以获得边界点的坐标了。这部分的具体代码如下：

```
std::vector<float> edge_lengths;
float perimeter=0.0f;
for(std::size_t i=0;i<boundary_vertices.size();++i)
{
    auto v_curr=boundary_vertices[i];
    auto v_next=boundary_vertices[(i+1)%boundary_vertices.size()];
    float length=glm::length(input.Positions[v_curr]-input.Positions[v_next]);
    edge_lengths.push_back(length);
    perimeter+=length;
}
float cumulative_length=0.0f;
for(std::size_t i=0;i<boundary_vertices.size();++i)
{
    auto v_idx=boundary_vertices[i];
    float t=cumulative_length/perimeter;
    float angle=t*2.0f*glm::pi<float>();
    cumulative_length+=edge_lengths[i];

 output.TexCoords[v_idx]=glm::vec2(0.5f+0.5f*std::cos(angle),0.5f+0.5f*std::sin(a
ngle));
}
```

设置完边界条件后，我们要进行Gauss-Seidel 迭代。我们先看原始公式：

$$x_i^{(new)} = (1/a_{ii}) * (b_i - \Sigma_{j<i} a_{ij} * x_j^{(new)} - \Sigma_{j>i} a_{ij} * x_j^{(old)})$$

在我们的Case中，

- $a_{ii} = d_i$
- $a_{ij} = -1$（如果 `j` 是`i`的邻居）
- $b_i = \Sigma_{k \in N_bdy(i)} uv_k$

代入后得到

$$uv_i^{(new)} = (1/d_i) * ((\Sigma_{k \in N_bdy(i)} uv_k) - \Sigma_{j<i, j \in N(i)} (-1) * uv_j^{(new)} - \Sigma_{j>i, j \in N(i)} (-1) * uv_j^{(old)})$$

也即

$$uv_i^{(new)} = (1/d_i) * ((\Sigma_{k \in N_bdy(i)} uv_k) + (\Sigma_{ji, j \in N(i)} uv_j^{(old)}))$$

于是我们得到

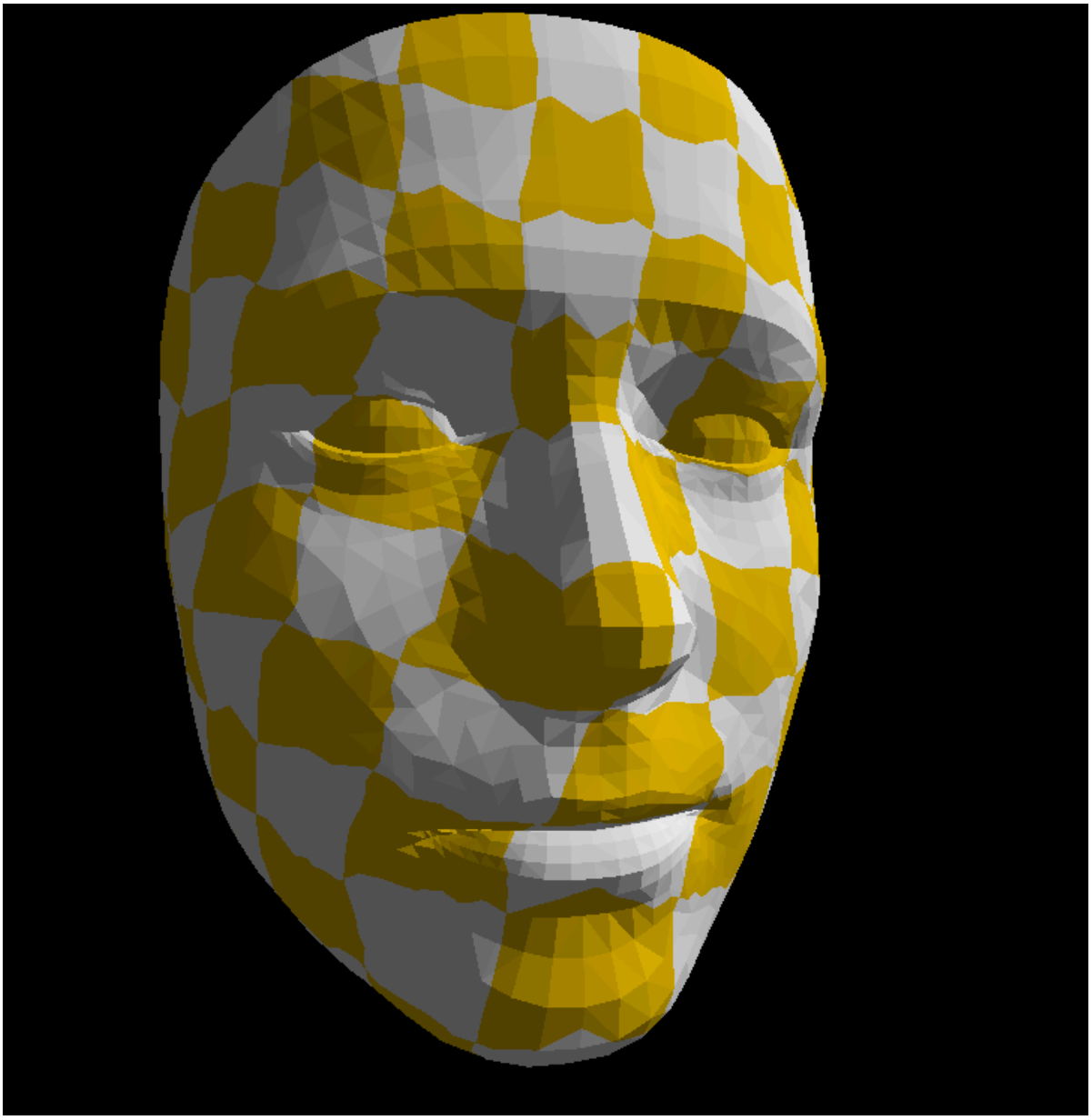
- 第一部分是所有边界邻居的UV坐标之和。
- 第二部分是所有已经更新过（在本次迭代中）的内部邻居的UV坐标之和。
- 第三部分是所有尚未更新（仍是上次迭代结果）的内部邻居的UV坐标之和。

把这三部分加在一起，就是所有邻居**（无论边界还是内部，无论新旧）**的UV坐标之和，所以，这个公式最终简化为了：

$$v_i^{(new)} = \frac{1}{d_i} \Sigma_{j \in N(i)} uv_j$$

所以在实现上，我们只需要每次根据顶点的索引值依次更新，每次都取其邻居的平均值即可，这些邻居有些是已经被更新过的，有些没有，这正是我们迭代的方法。

效果图如下：



可以观察到，UV 坐标收敛，并且相对均匀地分布在人脸上。

Task3

为了对每个顶点计算代价矩阵Q_p，我们需要先求出每个面的代价矩阵K_p。给定一个三角形面法向量$\vec{n} = (a, b, c)$，**需要先正规化使** $\| \vec{n} \| = 1$，则平面方程可以写为：$ax + by + cz + d = 0$，

把向量写成齐次形式$p = (a, b, c, d)^T$，那么面对应的二次代价矩阵K_p（4×4）定义为

$$K_p = pp^T = \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix}.$$

这个矩阵表示点 $v=(x,y,z,1)^T$ **（这里采用齐次坐标）**到该平面的平方代价（即距离）为 $v^TK_pv=(ax+by+cz+d)^2$。（在原始的距离公式中还有分母项$\sqrt{a^2+b^2+c^2}$，但由于我们在一开始正规化了$\vec{n}$，因此这一项被削去了）为了计算K_p，我们就需要得到平面的参数a,b,c,d，而确定平面的方向可以从其上的三个点来入手。将三角形面的两个边向量取出，并做叉乘和正规化得到平面的单位法向量：

```
auto v0 = f->VertexIndex(0);
auto v1 = f->VertexIndex(1);
auto v2 = f->VertexIndex(2);
glm::vec3 p0 = output.Positions[v0];
glm::vec3 p1 = output.Positions[v1];
glm::vec3 p2 = output.Positions[v2];
glm::vec3 normal = glm::normalize(glm::cross(p1 - p0, p2 - p0));
```

此时的`normal`向量便代表了平面的单位法向量，也就是我们想要的向量(a,b,c)。平面的方程于是表示为$\vec{n}\cdot(p-p_0)=0$，也即$x_nx+y_ny+z_nz-\vec{n}\cdot p_0=0$，因此我们要求的向量$p$就应该是 $p=(a,b,c,d)^T=(a,b,c,-\vec{n}\cdot p_0)$，在这种情况下平面方程就化为$p\cdot v=0$所以代码实现就是：

```
float d = -glm::dot(normal, p0);
glm::vec4 plane_eq(normal, d);
```

故$K_p=pp^T$=`glm::outerProduct(plane_eq,plane_eq)`。（`glm::outerProduct(c,r)`不是数学上的外积，而是将两个列向量做cr^T的矩阵乘法）

计算完每个面的K_p之后，每个点的二次代价矩阵Q_p相当于就是含有这个点的所有面的K_p的加和。在给出的代码中已经实现。

下面阐述求解最优坍缩位置$\bar{v}$的方法。翻阅论文，作者通过引用文献，阐述了求解v^TQv最小值**（v为齐次坐标下的点）**的方法相当于把Q的最后一行置为$(0\quad 0\quad 0\quad 1)$，同时右边变为$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$。这里在数学上实际上是等价的，将原矩阵分块，向量$v$也分块即可得到。这样做还天然保证了齐次坐标的最后一维一定是1，正好可以解决我们的约束。我们设改变后的矩阵为$\bar{Q}$。那么在其可逆的情况下，我们最优坐标的求解和对应代价就是：

```
glm::mat4 Qbar_inv = glm::inverse(Qbar);
Qbar[0][3] = 0.0f;
Qbar[1][3] = 0.0f;
Qbar[2][3] = 0.0f;
Qbar[3][3] = 1.0f;
cost= glm::dot(target_position, Q * target_position);
```

这里有一个实现上的细节，那就是`mat4`的行列和普通数组**相反**，所以最后一行实际上是所有第二坐标为3，因此只能逐个改动，`Qbar[3] = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);`**是错误的！**

论文中对于$\bar{Q}$不可逆的情况是这么阐述的：

> If this matrix is not invertible, we attempt to find the optimal vertex along the segment v1v2. If this also fails, we fall back on choosing $\bar{v}$ from amongst the endpoints and the midpoint.

这里为了实现简单，直接考虑作者提到的最后一种情况：直接比较3个候选点的`cost`，从中挑出最小的作为最终代价并把这个点的位置提交。**别忘了归一化！**最后`return ContractionPair{ edge, target_position, cost };`即可。

当我们坍缩合并了两个点之后，`v2`消失了，`v1`得到了保留，但是其坐标也发生了变化，因此需要更新`Qv`和`Kv`数据。为此我们需要重新计算和`v1`相邻的所有面以及这些面上所有点的`Qv`和`Kv`。我们沿着`v1`为中心的一个环走：这将取出所有含有`v1`的三角形的`v1`的对边，以及它们对应的面。代码根据hint中给出的步骤按部就班，可以看到实现和hint对应的很好：

```
Qv[v1] = glm::mat4(0);
for (auto e : ring) {
    // your code here:
    //     1. Compute the new Kp matrix for $e->Face()$.
    auto f = e->Face();
    auto faceidx=G.IndexOf(f);
    auto old_Kp = Kf[faceidx];
    auto new_Kp = UpdateQ(f);
    //     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 $Qv$).
    auto delta_Kp = new_Kp - old_Kp;
    auto i0= f->VertexIndex(0);
    auto i1= f->VertexIndex(1);
    auto i2= f->VertexIndex(2);
    if (i0 == v1)Qv[i0] += new_Kp;
    else Qv[i0] += (new_Kp - old_Kp);
    if (i1 == v1)Qv[i1] += new_Kp;
    else Qv[i1] += (new_Kp - old_Kp);
    if (i2 == v1)Qv[i2] += new_Kp;
    else Qv[i2] += (new_Kp - old_Kp);
    //     4. Update $Kf$.
    Kf[faceidx] = new_Kp;
}
```

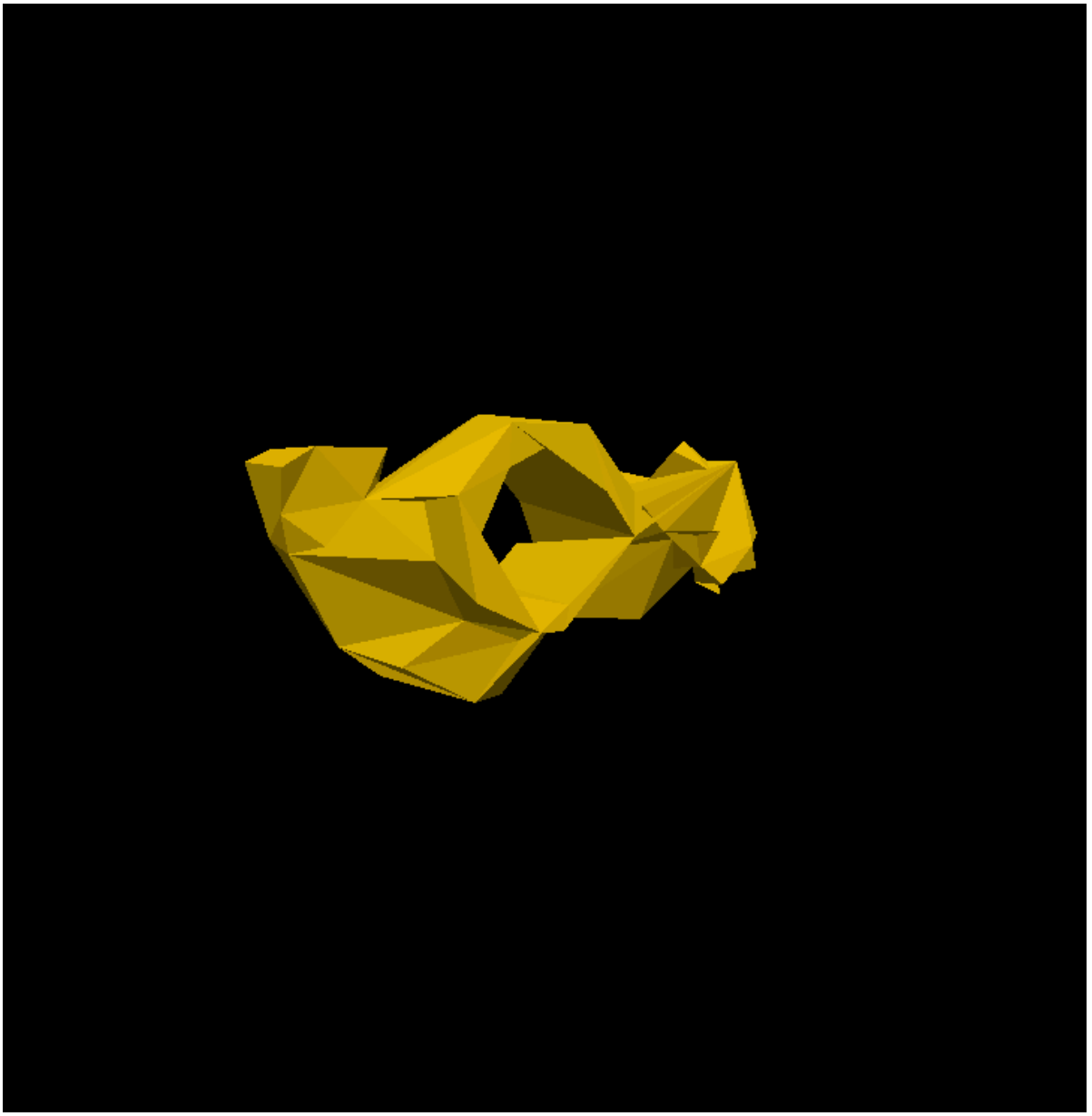
这里为了保证不出错，稳妥起见，并没有用边的`e->From()` `e->To()`等等函数来表示点，而是将它们全部一视同仁取出后再逐个判断它们是不是`v1`。对于`v1`，因为在循环外它的`Qv`被置零了，因此需要**直接**赋值`new_Kp`，其它的则需要在原有的基础上做修正`Qv+=(new_Kp - old_Kp)`。

最后，正如hint给出的，由于`Q`已经改变了，因此我们也需要在`ContractionPair`中更新每一个顶点对的代价。这一步可能要做两遍，因为它储存的是半边，如果这条半边有`twin`，那么同样需要更新。更新时，将`Qv`更新成`Qv[v1] + Qv[v2]`就可以了。具体实现形如：

```
for (auto e : ring) {
    auto edge_idx = G.IndexOf(e);
    if (pair_map.find(edge_idx) != pair_map.end()) {
        auto & pair = pairs[pair_map[edge_idx]];
        if (pair.edge) {
            auto v1 = pair.edge->From();
            auto v2 = pair.edge->To();
            pair = MakePair(pair.edge, output.Positions[v1],
output.Positions[v2], Qv[v1] + Qv[v2]);
        }
```

```
    }
    auto twin = e->TwinEdgeOr(nullptr);
    if (twin) {
        auto twin_edge_idx = G.IndexOf(twin);
        if (pair_map.find(twin_edge_idx) != pair_map.end()) {
            auto & pair = pairs[pair_map[twin_edge_idx]];
            if (pair.edge) {
                auto v1 = pair.edge->From();
                auto v2 = pair.edge->To();
                pair =
MakePair(pair.edge,output.Positions[v1],output.Positions[v2], Qv[v1]+Qv[v2]);
            }
        }
    }
}
```

此处仅展示rocker，剩余图像请见images文件夹。效果图如下：



可以看到，效果和tutorial给出的参考图几乎一致，保留了大体框架但是面数大大减少了。

Task4

给定顶点获取余切值的函数实现十分简单，这里略去不讲。

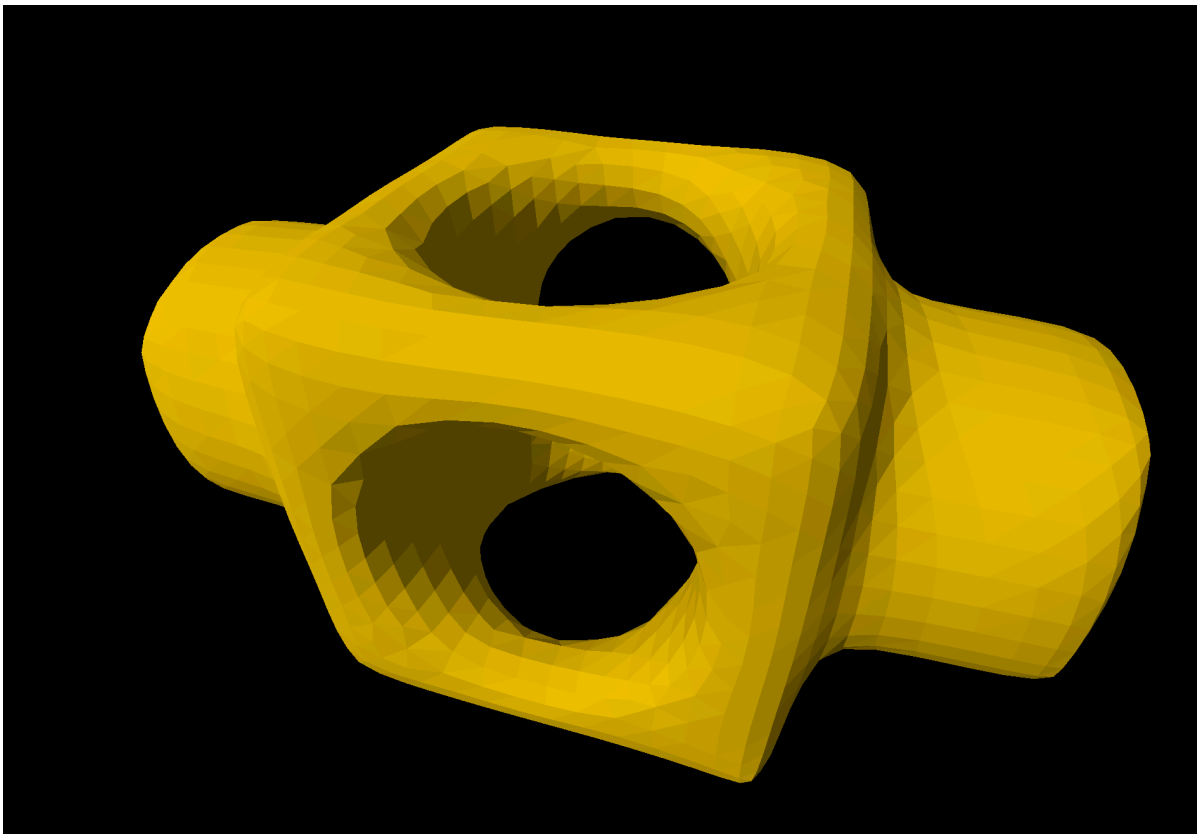
在更新顶点值的时候，对于余切权重我们需要找出当前顶点的一条边**两侧**的两个三角形。由于我们没有直接构造半边的函数，因此选用的外层循环只能是当前点`v=G.Vertex(i)`的`v->Ring()`，这样来遍历所有的环半边，进而按顺序取出我们需要的顶点。具体来说是这样的：

```
for(auto e:v->Ring())
{
    auto vj_idx=e->To();
    if(useUniformWeight)
    {
        laplacian+=prev_mesh.Positions[vj_idx];
        weight_sum+=1.0f;
    }
    else
    {
        auto v_alpha_idx=e->From();
        auto v_beta_idx=e->NextEdge()->TwinEdge()->NextEdge()->To();
        glm::vec3 v_alpha=prev_mesh.Positions[v_alpha_idx];
        glm::vec3 v_beta=prev_mesh.Positions[v_beta_idx];
        glm::vec3 vi=prev_mesh.Positions[i];
        glm::vec3 vj=prev_mesh.Positions[vj_idx];
        float cot_alpha=GetCotangent(v_alpha,vi,vj);
        float cot_beta=GetCotangent(v_beta,vi,vj);
        laplacian+=(cot_alpha+cot_beta)*prev_mesh.Positions[vj_idx];
        weight_sum+=(cot_alpha+cot_beta);
    }
}
```

对于均匀权重，我们只需要对每条环边取出它的头（或尾，均可）即可。对于余切权重，我们取`vj_idx=e->To()`来获得我们要计算的v_j的权重，因此现在处理的这条边`e`是以`glm::vec3 vj=prev_mesh.Positions[vj_idx];`为终点的，那么我们的α_{ij}角的顶点的索引也就是`v_alpha_idx=e->From();`了。此时，βij角所在的顶点是与`e`所在三角形共享边v_iv_j的另一三角形的、边v_iv_j的相对的顶点。为此我们需要含有这个点的一条半边，我们做`v_beta_idx=e->NextEdge()->TwinEdge()->NextEdge()->To();`，根据半边数据结构的性质，我们可以知道这样得出的点确实就是β_{ij}的顶点。然后执行常规的权重计算和位置更新即可。

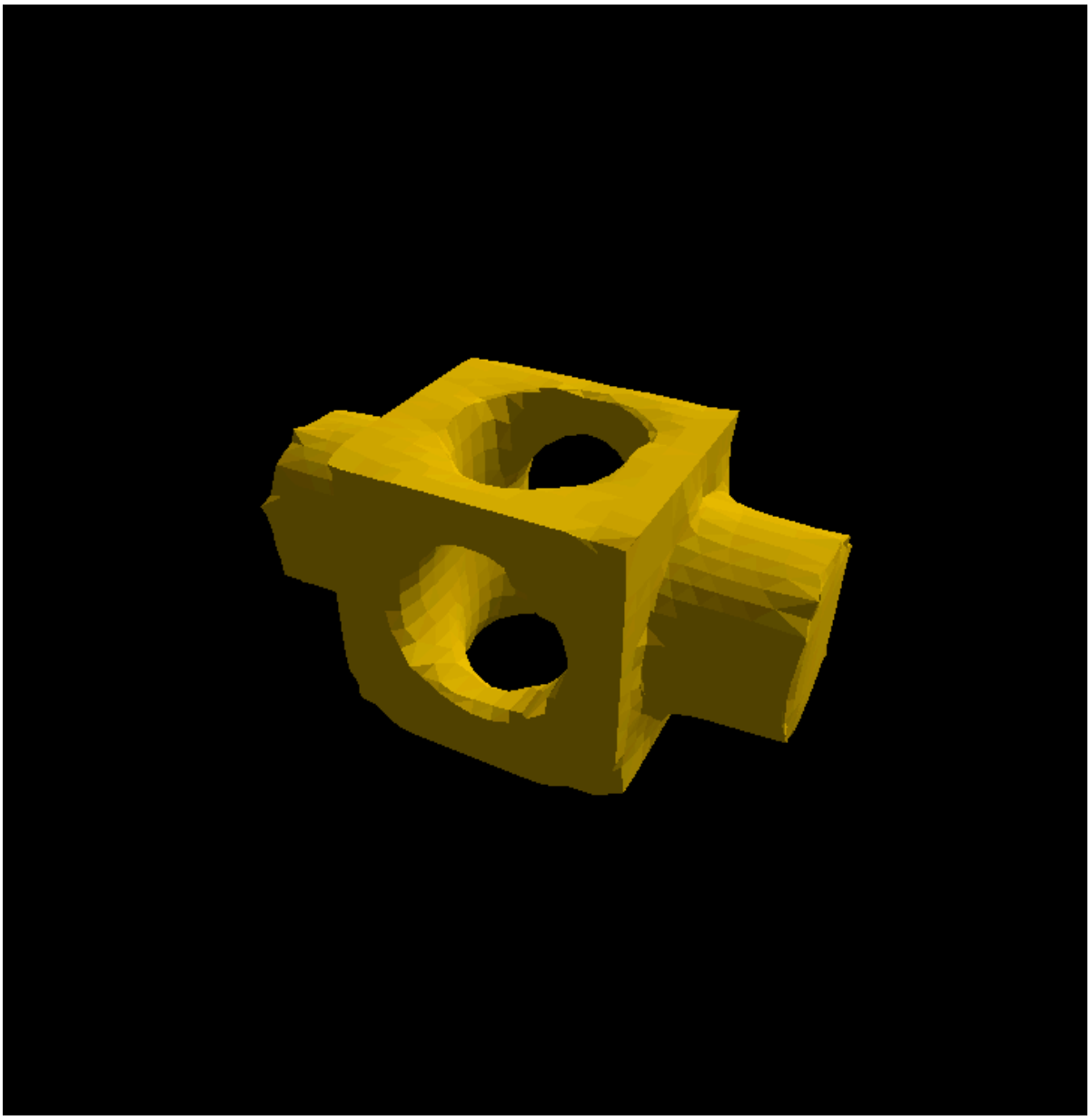
此处没有对余切权重为负的情况做特殊处理，但是对任何除以零的地方都做了检查。

`block.obj`在$\lambda = 0.5$，`numIterations=10,useUniforWeight=True`的情况下的效果图如下：



观察到物体的表面没有了棱角，也变得光滑细腻了很多，与tutorial中给出的参考图基本一致。

block.obj在$\lambda = 0.5$，`numIterations=10,useUniforWeight=False`的情况下（也即使用Cotangent权重）的效果图如下：



观察到相比uniform权重，cotangent权重对棱角的保留更好，其他地方也能做得很好的平滑。

Task5

这一部分没有给任何的模板，所以主观上实现难度比前面的几个Task要高很多。

hint中给出了如下的参考步骤：

> 1. 为网格结构的边建立查询其上有无 mesh 顶点的数据结构；
> 2. 逐网格判断哪些边上有 mesh 的顶点；
> 3. 逐网格连接相应边上 mesh 的顶点，组成三角 mesh。

那么我们就按部就班地按照这些步骤来一一实现。

首先是建立联系边和有无顶点的数据结构。这里采用了一个四维数组来存储：

```
std::vector<std::vector<std::vector<std::array<int,3>>>>point_idx(n+1,std::vector
<std::vector<std::array<int,3>>>(n+1,std::vector<std::array<int,3>>(n+1,
{-1,-1,-1})));
```

（这里不能直接使用`int point_idx[n+1][n+1][n+1][3];`，因为当Resolution变大时，`n`会急剧变大导致爆内存，所以需要使用动态数组`vector`）

这个数组的意思是，`point_idx[nx][ny][nz][i]=j`代表从坐标为`glm::vec3{dx*nx,dy*ny,dz*nz}+grid_min`的点出发，方向为`unit(i)`的那条边上存在编号为`j`的点。（-1代表该点不存在）这样虽然空间开销比较大，但是查询的复杂度是O(1)。而且实现比较简单，如果使用`std::unordered_map`的话，需要为`glm::vec3`实现哈希函数，而这是库中没有提供的；如果使用`std::map`，由于`glm::vec3`没有提供严格弱序（运算符`<`未定义），因此需要重载小于号，也是比较繁琐的。因此使用数组就好。

其次是逐网格判断哪些边上有mesh顶点。用三重循环来遍历整个网格。对每个网格，我们做如下的事情：

```
std::vector<glm::vec3> cube_vertices(8);
std::vector<float> cube_sdf(8);
std::vector<int> edge_vertex_idx(12,-1);
cube_vertices[0]=grid_min+glm::vec3(nx*dx,ny*dx,nz*dx);
for(int i=0;i<8;++i)
    cube_vertices[i]=grid_min+glm::vec3((nx+(i&1))*dx,(ny+((i>>1)&1))*dx,(nz+
(i>>2))*dx);
for(int i=0;i<8;++i)
    cube_sdf[i]=sdf(cube_vertices[i]);
int v_binary_index=0;
for(int i=7;i>=0;--i)
{
    if(cube_sdf[i]>0.0f)
        v_binary_index++;
    v_binary_index<<=1;
}
v_binary_index>>=1;
int edge_flags=c_EdgeStateTable[v_binary_index];
if(edge_flags==0)continue;
for(int i=0;i<12;++i)
{
    int unit_num[]={0,0,0};
    unit_num[((i>>2)+1)%3]=i&1;
    unit_num[((i>>2)+2)%3]=(i>>1)&1;
    if(edge_flags&(1<<i))
    {
        glm::vec3 pstart,pend,pnew;
        if(point_idx[nx+unit_num[0]][ny+unit_num[1]][nz+unit_num[2]][i>>2]==-1)
        {
            point_idx[nx+unit_num[0]][ny+unit_num[1]][nz+unit_num[2]]
[i>>2]=output.Positions.size();

 pstart=dx*glm::vec3{unit_num[0],unit_num[1],unit_num[2]}+cube_vertices[0];
            pend=pstart+unit[(i>>2)]*dx;
            float sdf_start=sdf(pstart);
            float sdf_end=sdf(pend);
            float t=std::abs(sdf_start)/(std::abs(sdf_start-sdf_end));
            pnew=(1.0f-t)*pstart+t*pend;
            output.Positions.push_back(pnew);
            edge_vertex_idx[i]=output.Positions.size()-1;
        }
```

```
        else edge_vertex_idx[i]=point_idx[nx+unit_num[0]][ny+unit_num[1]]
[nz+unit_num[2]][i>>2];
    }
}
```

`cube_vertices`、`cube_sdf`数组储存的是每个立方体中点的具体坐标和sdf值，其下标和约定的顺序一致。`edge_vertex_idx`数组则是储存边上的mesh顶点的全局编号，具体来说，`edge_vertex_idx[i]=j`就代表当前立方体的边e_i上有全局编号为`j`的mesh顶点。（-1为空）

接着我们计算这个立方体的8位二进制数`v`，在这里命名为`v_binary_index`。具体来说也就是对每个顶点，第 `i` 位为 1 或 0 分别表示在立方体的第 `i` 个顶点处 sdf 为正或负；我们只需要做一次循环构建即可，具体见代码。得到了`v`以后我们可以得到12位二进制数`edge_flags`，它的第`i`位代表第`i`条边上是否有mesh顶点。我们逐位考虑，如果这条边上的顶点没有被定义过，那么首先需要在`point_idx`数组中标记一下。随后，由于第 `j` 条边的起始点为 ：`v0+dx*(j&1)*unit(((j>>2)+1)%3)+dx*((j>>1)&1)*unit(((j>>2)+2)%3)`，因此我们定义`unit_num`来存储当前边的起点相对$v0$的位置用三个基矢分解后所得系数，进而得到边的起点的真实坐标`pstart=dx*glm::vec3{unit_num[0],unit_num[1],unit_num[2]}+cube_vertices[0];`

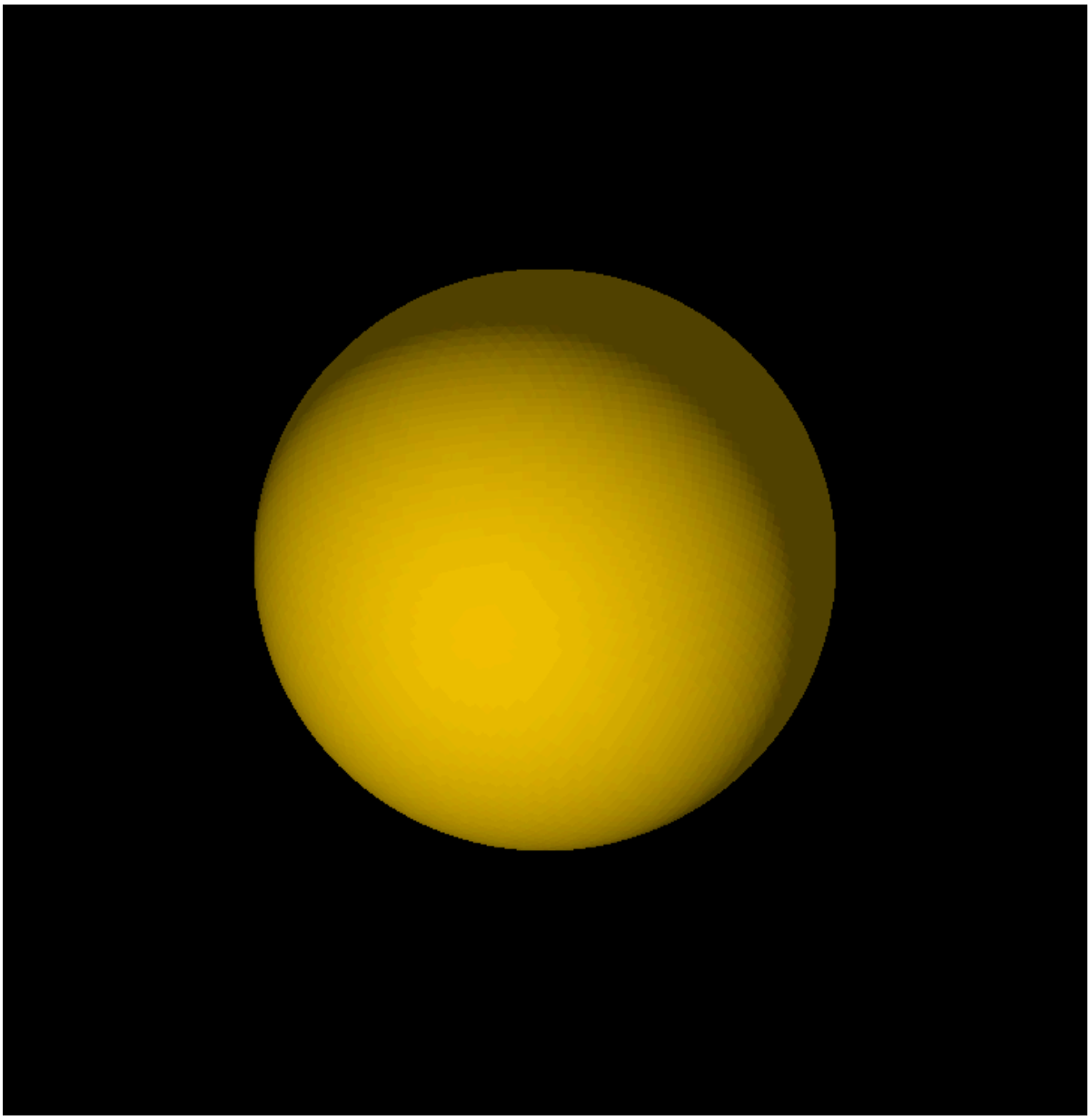
由于边的方向是`unit(j>>2)`，所以`pend=pstart+unit[(i>>2)]*dx;`就是终点的真实坐标。得到这两个坐标后，我们通过插值公式$p^\star=\frac{v_2-v^\star}{v_2-v_1}p_1+\frac{v^\star-v_1}{v_2-v_1}p_2$来得到最终mesh顶点的真实坐标`pnew`。这个Case中我们认为等值面$v^\star=0$，所以公式被简化为$p^\star=\frac{v_2p_1-v_1p_2}{v_2-v_1}$，可以看到与代码中的实现一致。最后我们需要把计算出的点添加到`output.Positions`中。

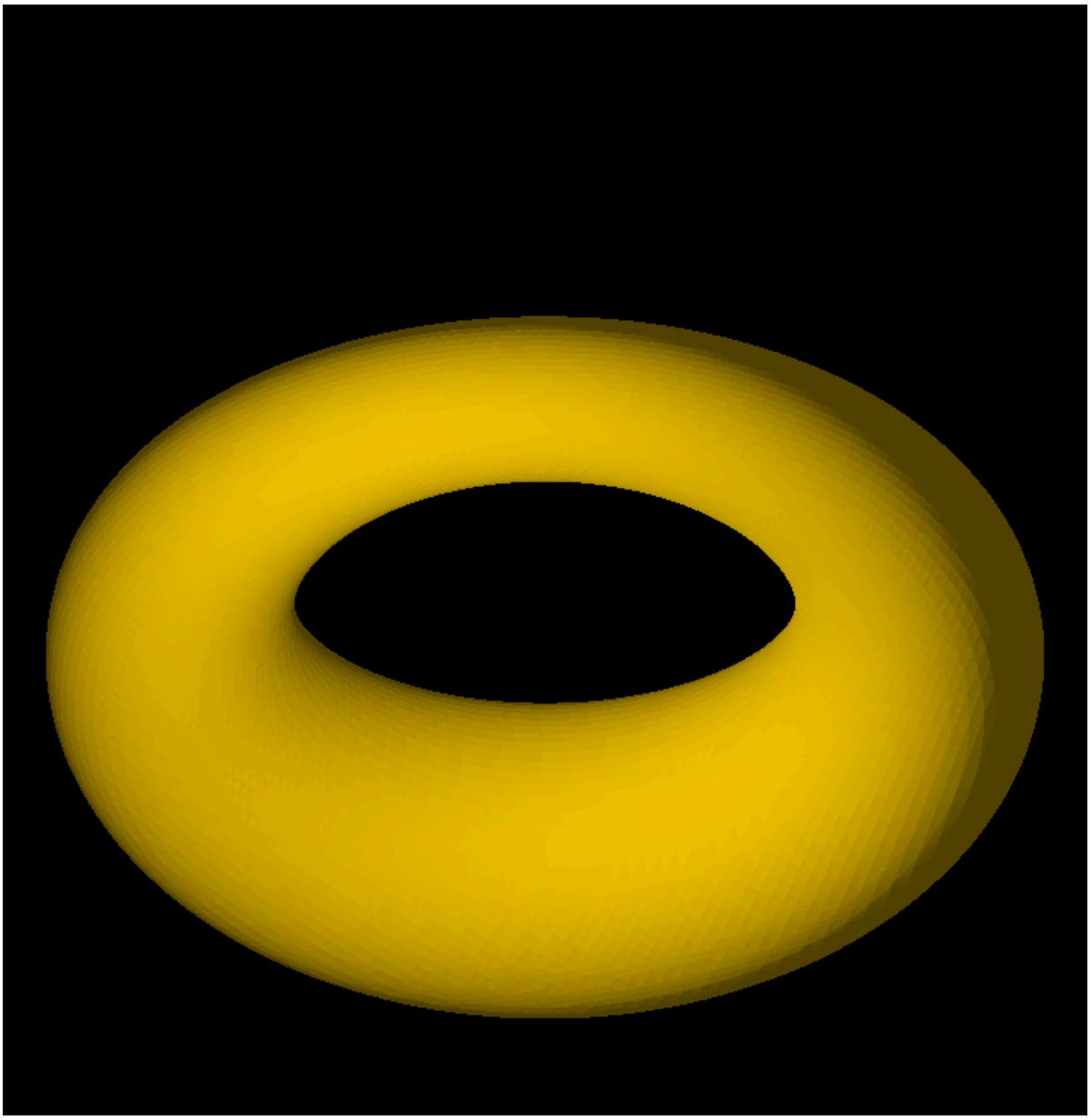
最后，我们把这些点连成面。我们在前面的处理中，可以同时标记边e_i上顶点的全局下标（序号），也就是代码中`edge_vertex_idx[i]=output.Positions.size()-1;`的一句（对于新添加的顶点），或是`edge_vertex_idx[i]=point_idx[nx+unit_num[0]][ny+unit_num[1]][nz+unit_num[2]][i>>2];`一句（对于已添加的顶点），注意这里即使一个顶点已经被添加过了，它在这个立方体中的`edge_vertex_idx`数组中**依旧需要重新登记**，否则它无法被我们当前的立方体识别。得到了哪些边上有点以及它们的下标之后，我们只需要通过`c_EdgeOrdsTable`数组储存的信息来把对应的点连起来即可。

```
for(int i=0;i<5;++i)
{
    uint32_t e0=c_EdgeOrdsTable[v_binary_index][3*i];
    uint32_t e1=c_EdgeOrdsTable[v_binary_index][3*i+1];
    uint32_t e2=c_EdgeOrdsTable[v_binary_index][3*i+2];
    if(e0==-1)break;
    output.Indices.push_back(edge_vertex_idx[e0]);
    output.Indices.push_back(edge_vertex_idx[e1]);
    output.Indices.push_back(edge_vertex_idx[e2]);
}
```

注意这里`e0==-1`就退出循环的原因是在`c_EdgeOrdsTable`数组中，前面的元素全不为-1，第一个-1之后的元素全为-1，且非-1的元素个数一定为3的倍数（这是由其定义决定的），因此它就是存在与不存在的分界线。当我们遇到第一个-1元素的时候，说明当前立方体的顶点已经全部处理完毕，因此可以退出循环。一个立方体中最多有4个三角形mesh，因此我们需要做4遍循环，每次取出3个点连成面。

最终实现的效果图如下：





（两者均为Resolution=100的情况）

可以看到这样构建出来的圆球和圆环很光滑，重建很成功。