Lab2 Report

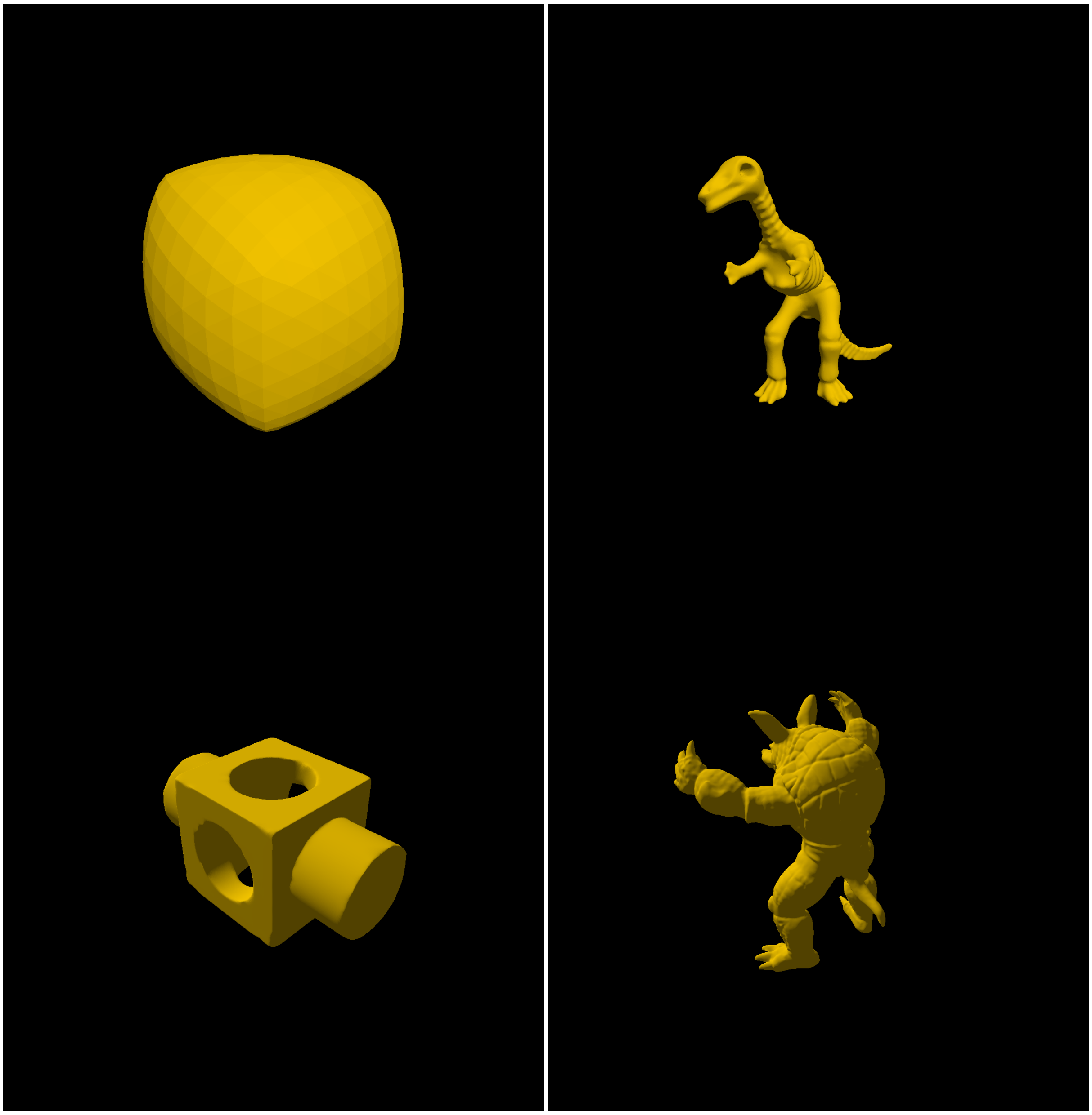
Task 1: Loop Mesh Subdivision

在这个任务中，我实现了三角网格细分算法。每一轮细分操作，在三角形的每一条边上产生一个顶点，并且对原有的顶点重新调整。因此，在实现过程中，对于每一轮细分操作，我分成了四步：

- 建立查询边的数据结构
- 计算原有的顶点的位置
- 计算新生成的顶点位置
- 建立顶点之间的连接关系
  在第一步中，我使用助教提供的`DCEL`数据结构来查询边。在第二步中，遍历每一个顶点，根据周围顶点的位置和该定点原来的位置来计算这个顶点的新位置,并且按照周围顶点权重 $\frac{3}{8n}$ ，该顶点权重 $\frac{5}{8}$ 进行加权。在第三步中，遍历每一条边，根据论文中的公式进行加权计算出新加入的顶点的位置，并且记录每一边所加入的顶点的索引。在这里，我定义了一个宏。第四步中，遍历每一个三角形，根据之前保存的索引，将新的四个三角形加入到新的网格中，在这里需要注意加入点的顺序。

```
// Macro
#define MAP_PAIR(a, b) ((((uint64_t) a + b) << 32) + abs((long long) (a -  b)))
// Map the edge (v2,v3) to the index of new vertex
map_record[MAP_PAIR(v2, v3)] = New.Positions.size() - 1;
```

最终的结果如下图所示：

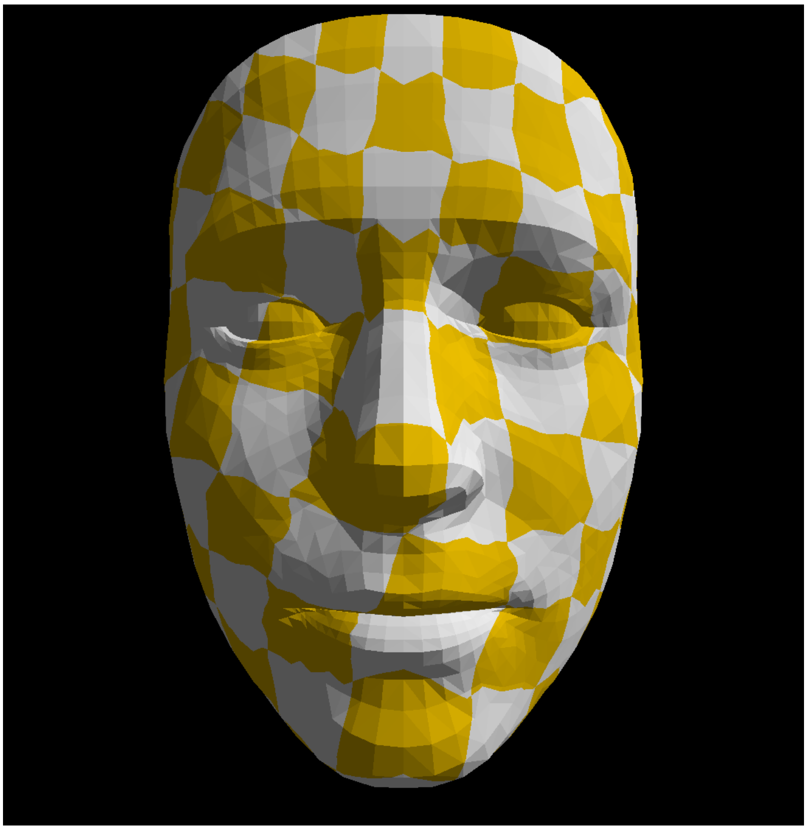


Task 2: Spring-Mass Mesh Parameterization

在这个任务中，我实现了基于弹簧质点的三角网格参数化算法。首先，我使用助教提供的DCEL数据结构来查询边。然后，通过循环，我找到了一个边界点，以这个边界点为起点，按照顺序找出所有的边界点。而后，我将边界点上的(u, v)坐标初始化为圆边界，其它点初始化为圆心附近的点。最后，我使用迭代法求解中间节点上的(u, v)坐标，为了方便，我直接使用$\lambda_{ij} = 1/n_i$ 的平均权重。

```
// Init side vertex
for (int i = 0; i < side_vertex.size(); i++) {
        output.TexCoords[side_vertex[i]] = glm::vec2(
            0.5 + 0.5 * sin(2 * PI * i / side_vertex.size()),
            0.5 + 0.5 * cos(2 * PI * i / side_vertex.size()));
}
// Iteration
texco.push_back(glm::vec2(0));
for (int j = 0; j < v_neighbors.size(); j++) {
    uint32_t u = v_neighbors[j];
    texco[i] = texco[i] + glm::vec2(1.0 / v_neighbors.size()) * output.TexCoords[u];
}
```

最终的结果如下图所示：



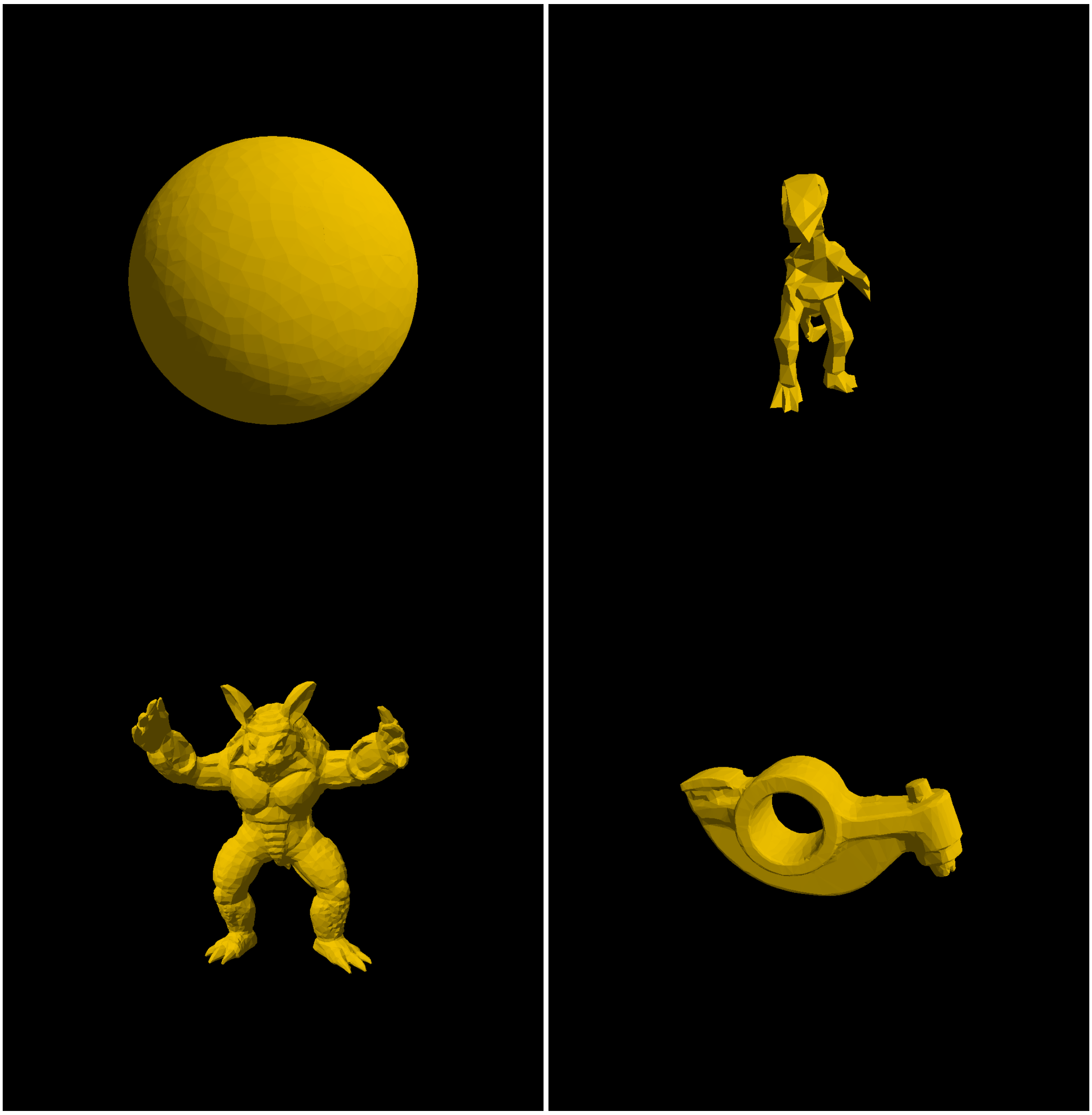
Task 3: Mesh Simplification

在这个任务中，我实现了网格简化的算法。首先，我对每一个初始的顶点计算二次代价矩阵，为了实现的方便，我定义了一个函数来计算平面的方程。然后使用助教提供的查询边的数据结构DCEL建立查询边的数据结构，遍历每一个顶点即可得到每一个顶点的二次代价矩阵。第二步是确定合法点对，和发电对主要包含两部分：有连边的顶点对和距离小于阈值的顶点对。我首先遍历所有的边，记录下这些点对，同时为了防止重复，我使用set来记录哪些点对加入了合法点对。然后遍历每一对点对，计算距离，将距离小于阈值，且未连边的点对加入合法点对。由于点对不区分顺序，所以在set中，我使用一个64位的数对点对进行编码MAP_PAIR(a, b) ((((uint64_t)a + b) << 32)

`+ abs( a -  b)))`。也就是`Task 1`中使用的宏。而后，对于每一对顶点我按照论文中的公式进行计算，并且记录最优收缩点和代价。最后每次都处理一对代价最小的点对，直到剩下的顶点数量小于要求的。在使用最优收缩点代替点对时，我会对网格和合法点对进行更新，删去退化的面、退化的合法点对，更新重新计算包含发生变化的点的代价和最优收缩点。

```
// calculate the equition of the plane.
glm::vec4 plane_equition(glm::vec3 v1, glm::vec3 v2, glm::vec3 v3) {
        glm::vec3 a = v1 - v2, b = v1 - v3;
        a = glm::vec3(
            a[1] * b[2] - a[2] * b[1], a[2] * b[0] - a[0] * b[2], a[0] * b[1] - a[1] * b[0]);
        b = a * a;
        if (b[0] + b[1] + b[2] == 0) return glm::vec4(1, 0, 0, 0);
        a = a / glm::vec3(sqrt(b[0] + b[1] + b[2]));
        b = a * v1;
        return glm::vec4(a[0], a[1], a[2], -b[0] - b[1] - b[2]);
    }
```

最终结果如下图所示：



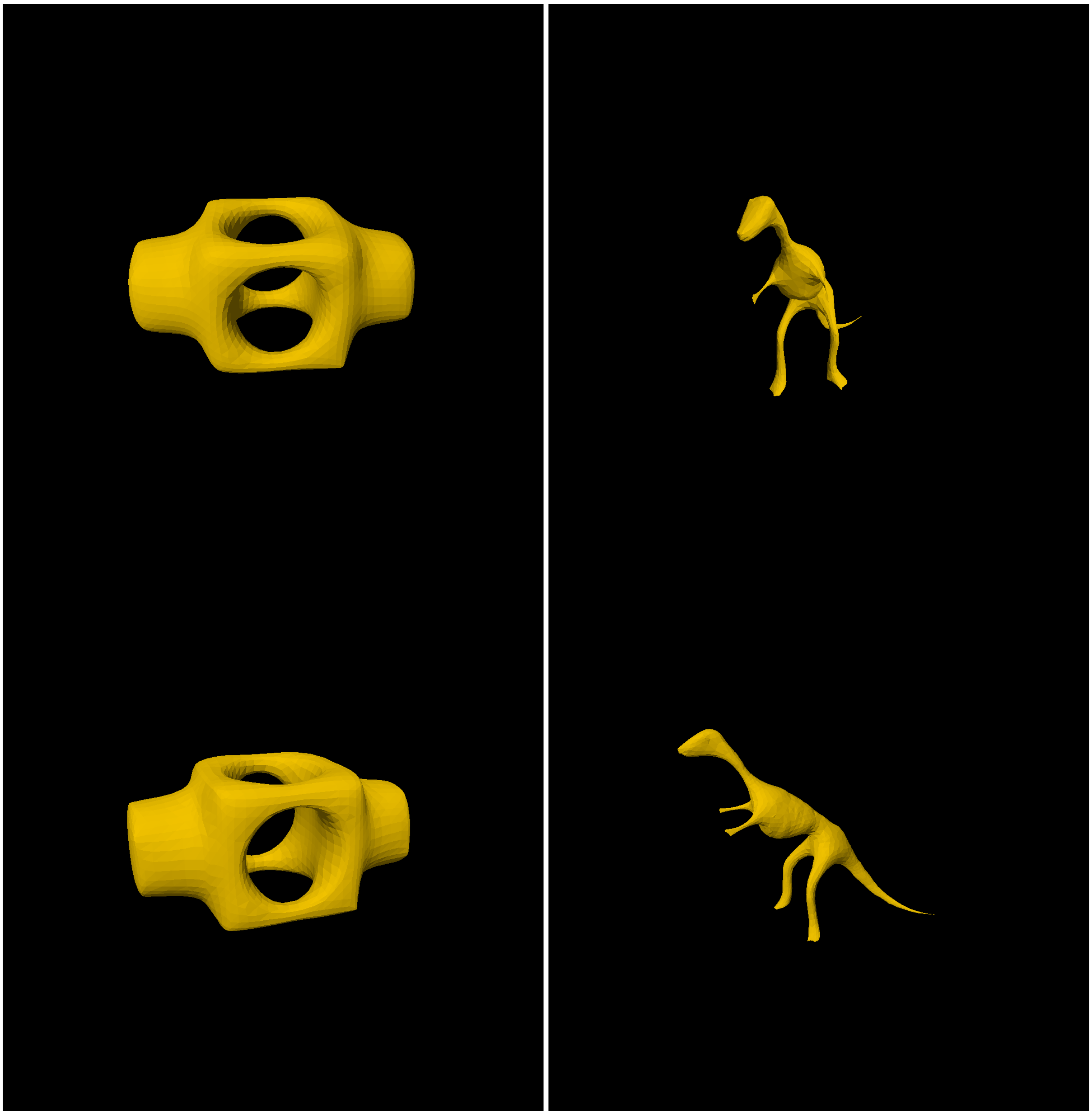
Task 4: Mesh Smoothing

在这个任务中，我实现了网格平滑算法。对于每一次迭代，我都进行如下步骤：

1. 遍历所有顶点，计算邻居位置的加权平均$v_i^* = \frac{\sum_{j\in N(i)} w_{ij}v_j}{\sum_{j\in N(i)} w_{ij}}$
2. 计算权值
3. 更新顶点：$v_i = (1-\lambda)v_i + \lambda v_i^*$

   为了实现的方便，我添加了一个函数`float my_cot(glm::vec3 v1, glm::vec3 v2, glm::vec3 v3)`。由于cot可能会出现负值，我将所有的cot都取绝对值作为权重。

最终结果如下图所示：



Task 5: Marching Cubes

在这个任务中，我实现了Marching Cube算法。为了实现的方便，我定义几个宏。NODE_MAP对正方体的点进行编码，保证在正方体构成的网格中每一个点都有唯一的编码，EDGE_MAP对正方体的边进行编码，保证在正方体构成的网格中每一个边都有唯一的编码，VERTEX_POSI给定v_0坐标，求v_i的坐标。我们遍历每一个正方体，利用助教提供的数据结构来判断隐式网格和正方体的交点情况。而后查看该交点之前是否已经加入网格，如果在之前的正方体中遍历到这个点，那么我们将它加入网格，然后我利用助教提供的表，进行连接，构建三角形。

```
// Macro
#define MAP_PAIR(a, b) \
    (((uint64_t) ((long long) a + b) << 32) + abs((long long) ((long) a - (long) b)))

#define NODE_MAP(a, b, c, t) \
    ((a + ((t >> 0) & 1)) * 100 * 100 * 4 + (b + ((t >> 1) & 1)) * 100 * 2 + c + ((t >> 2) & 1))

#define VERTEX_POSI(p, i) \
    (glm::vec3((p[0] + (i & 1) * dx), p[1] + ((i >> 1) & 1) * dx, p[2] + ((i >> 2) & 1) * dx))

#define EDGE_MAP(a, b, c, i, j) (MAP_PAIR(NODE_MAP(a, b, c, i), NODE_MAP(a, b, c, j)))
```