ACG - Report 1

张洁琼, 2022010603，计科 22

1 Mesh Subdivision

1.1 Data Structures

1.1.1 Edge Midpoint Map

- `std::map<Edge,int> edge_vertex_map`: This map stores the midpoint of each edge and its corresponding vertex index. It avoids recalculating midpoints for the same edge multiple times and ensures efficient retrieval during subdivision.

1.1.2 Vertex Index Map

- `std::map<glm::vec3,int,vec3compare> index_map`: This map holds the relationship between vertex positions (`glm::vec3`) and their corresponding indices in the vertex array. It allows for quick lookups to determine whether a vertex has already been added, avoiding duplicate entries.

1.1.3 New Vertices and Faces

- `std::vector<glm::vec3> vertices_new` and `std::vector<glm::i32vec3> faces_new`: These vectors store the new vertices and faces generated after each subdivision iteration. At the end of each iteration, the mesh's vertex and face data are updated using these vectors.

1.2 Helper Functions

1.2.1 Vertex Index Retrieval

The function `get_vertex_index()` checks if a vertex already exists in `vertices_new`. If the vertex does not exist, it adds the vertex to `vertices_new` and updates `index_map` with the new index. If the vertex already exists, it simply returns the index of the vertex. This avoids adding duplicate vertices.

1.2.2 Inserting Triangles

The function `insert_triangle()` inserts a new triangle into the mesh. It retrieves the vertex indices for the three vertices using `get_vertex_index` and adds the triangle (represented by a tuple of vertex indices) to `faces_new`.

1.3 Core Subdivision Logic

1.3.1 Edge Midpoint Calculation

The function `add_edge_vertex` calculates and stores the midpoint of each edge. Given two vertices v_1 and v_2, the midpoint is computed and added to `vertices_new` using the following equation:mid_point $= \frac{1}{2}(\mathbf{v_1} + \mathbf{v_2})$. The position is not important here actually as it will be recalculate later in the "Edge Vertex Update" part. The index of the midpoint is then stored in `edge_vertex_map` to avoid duplicate calculations for the same edge.

1.3.2 Subdividing Faces

For each triangle in the original mesh, midpoints are computed for its three edges ($v_0 - v_1$, $v_1 - v_2$, and $v_2 - v_0$). These midpoints are used to create four new smaller triangles that replace the original one:

1. A triangle formed by v_0, the midpoint of $v_0 - v_1$, and the midpoint of $v_0 - v_2$.
2. A triangle formed by the midpoint of $v_0 - v_1$, v_1, and the midpoint of $v_1 - v_2$.
3. A triangle formed by the midpoint of $v_0 - v_2$, the midpoint of $v_1 - v_2$, and v_2.
4. A triangle formed by the three midpoints of the original triangle's edges.

This process increases the number of triangles, refining the mesh detail. And the position of midpoints is calculated in the "Edge Vertex Update" part.

1.4 Vertex Position Updates

1.4.1 Edge Vertex Update

For each midpoint on an edge, the algorithm adjusts its position based on adjacent vertices. If the edge is part of the interior of the mesh, the new position is calculated using the positions of the two endpoints and two vertices from the adjacent triangles using the Catmull-Clark method:

$$\text{new\_pos} = \frac{3}{8} \cdot (\mathbf{A} + \mathbf{B}) + \frac{1}{8} \cdot (\mathbf{C} + \mathbf{D})$$

If the edge is on the boundary, the midpoint is simply the average of the two endpoints.

1.4.2 Original Vertex Update

Each original vertex in the mesh is updated based on its adjacent vertices. For boundary vertices, a weighted average is used:

$$\text{new\_pos} = \frac{3}{4} \cdot \text{original\_pos} + \frac{1}{8} \cdot \sum \text{adjacent\_pos}$$

For non-boundary vertices, a smoothing factor β is computed using the Catmull-Clark formula:

$$\beta = \frac{5}{8} - \left(\frac{3}{8} + \frac{\cos\left(\frac{2\pi}{k}\right)}{4} \right)^2$$

Here, k is the number of adjacent vertices. The new position is calculated as:

$$\text{new\_pos} = (1 - k \cdot \beta) \cdot \text{original\_pos} + \beta \cdot \sum \text{adjacent\_pos}$$

1.5 Mesh Update

At the end of each iteration, the original vertices and faces are replaced by the new ones (`vertices_new` and `faces_new`). This ensures that the mesh becomes progressively refined with each subdivision iteration.

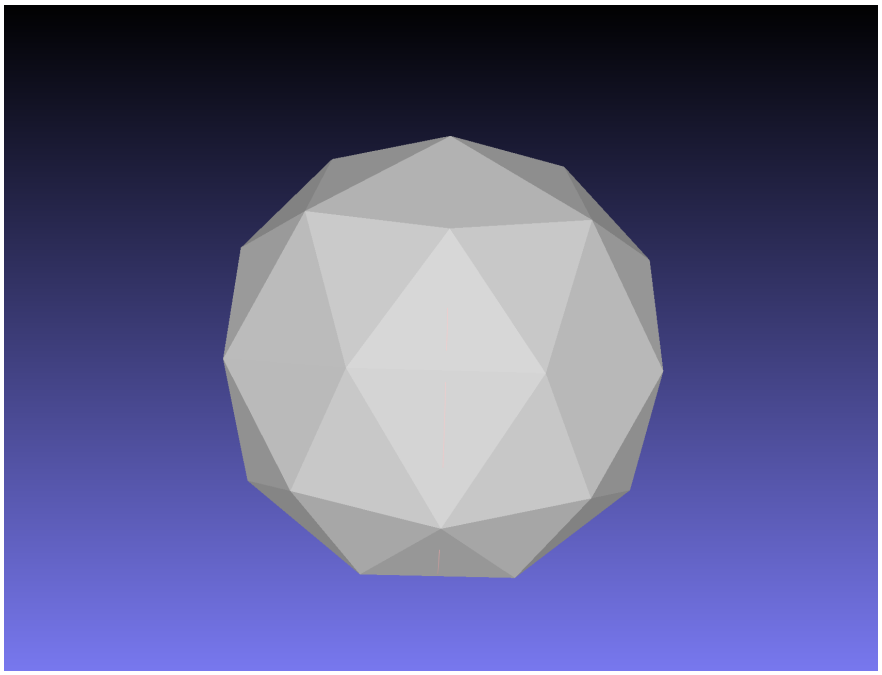

图 1: icosahedron_subdivided_1.obj subdivide 1

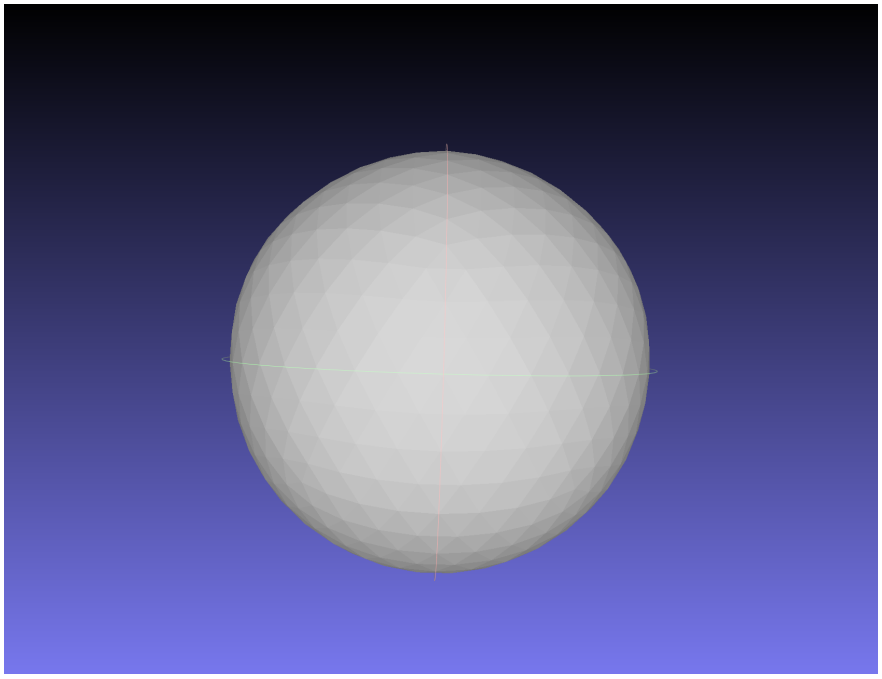

图 2: icosahedron_subdivided_3.obj subdivide 3

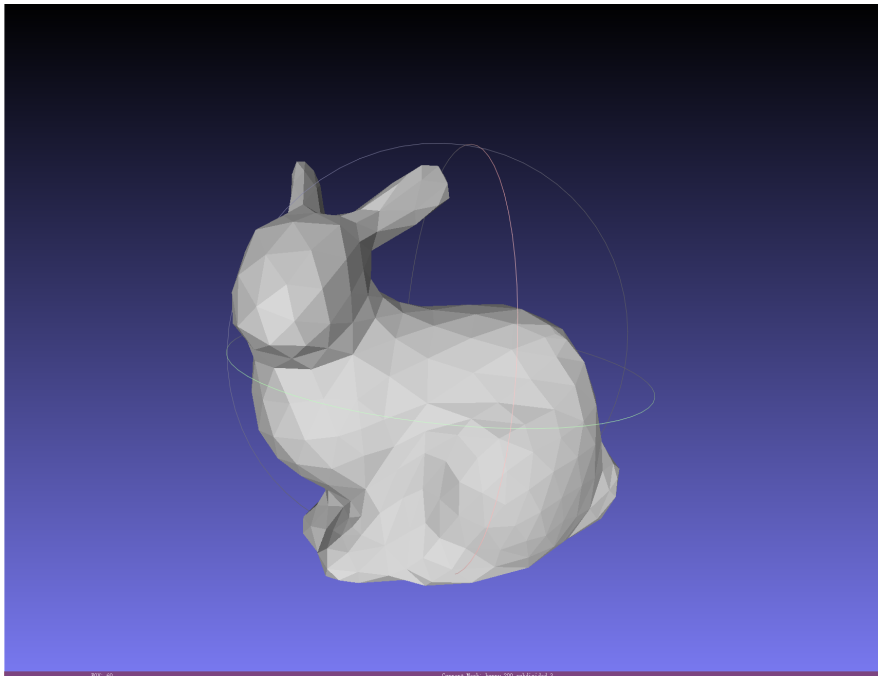

图 3: bunny_200_subdivided_1.obj subdivide 1

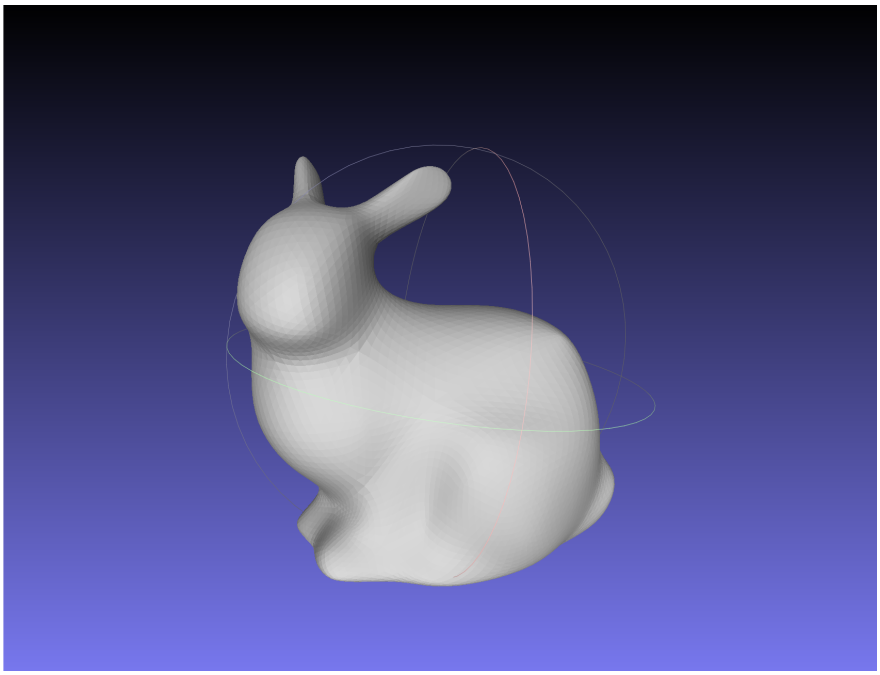

图 4: bunny_200_subdivided_3.obj subdivide 3

2 Mesh Simplify

2.1 QMatrix Function

2.1.1 Purpose

The `QMatrix` function computes the Q matrix associated with a triangle defined by three vertices. This matrix is used in mesh simplification to measure the quadric error related to the triangle and its vertex positions.

2.1.2 Code Details

1. Normal Vector Calculation:

Computes the normal vector of the triangle's plane using the cross product of two edge vectors. The result is normalized to ensure it has unit length.

2. Degenerate Triangle Check:

If the length of the normal vector is below a threshold (1e-4), the triangle is considered degenerate, and a zero matrix is returned.

3. Extracting Plane Equation Coefficients:

The components A, B, and C correspond to the coefficients of the plane equation $Ax+By+Cz+D=0$.

4. The D coefficient is calculated using the dot product of the normal vector and one of the triangle's vertices.

5. The Q matrix is constructed using the coefficients of the normal vector and D. This symmetric matrix encodes the error associated with points in relation to the triangle's plane.

2.2 GenerateMergedPoint Function

2.2.1 Purpose

The `GenerateMergedPoint` function generates an optimal merged point using the Q matrix. This point is derived from minimizing the quadric error defined by the Q matrix.

2.2.2 Code Details

1. A 3x3 matrix is extracted from the top-left portion of the Q matrix, which is used for solving the optimal point. Then, a vector b is constructed from the last column of the Q matrix.

2. Solving the Linear System:

The determinant of the 3x3 matrix is calculated. If the determinant is significantly non-zero, the inverse is computed to find the optimal merged point. If the matrix is nearly singular, a pseudo-inverse is used to ensure stability.

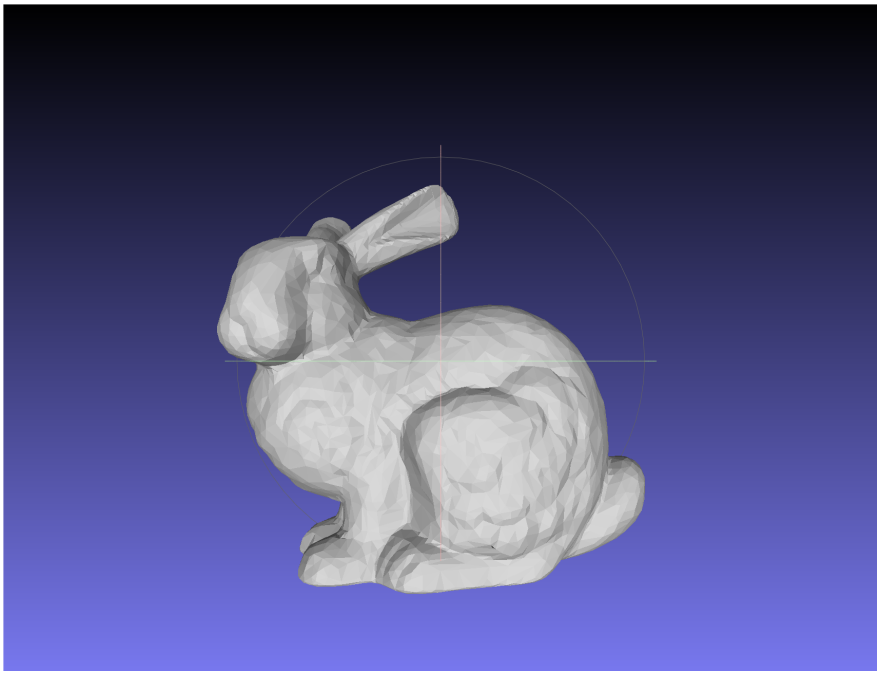

图 5: bunny_simplified_10000.obj simplify 10000

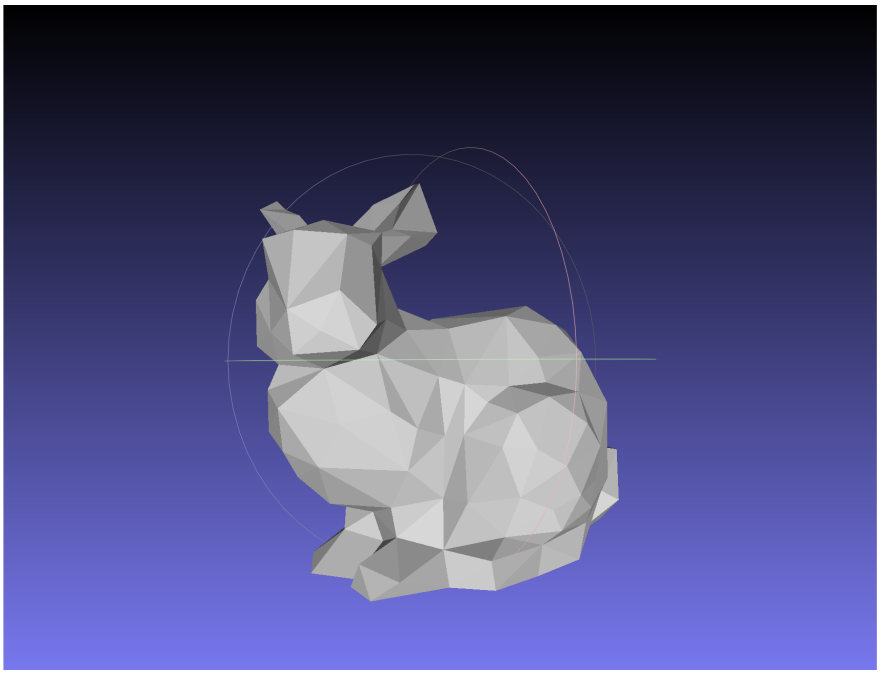

图 6: bunny_simplified_17000.obj simplify 17000

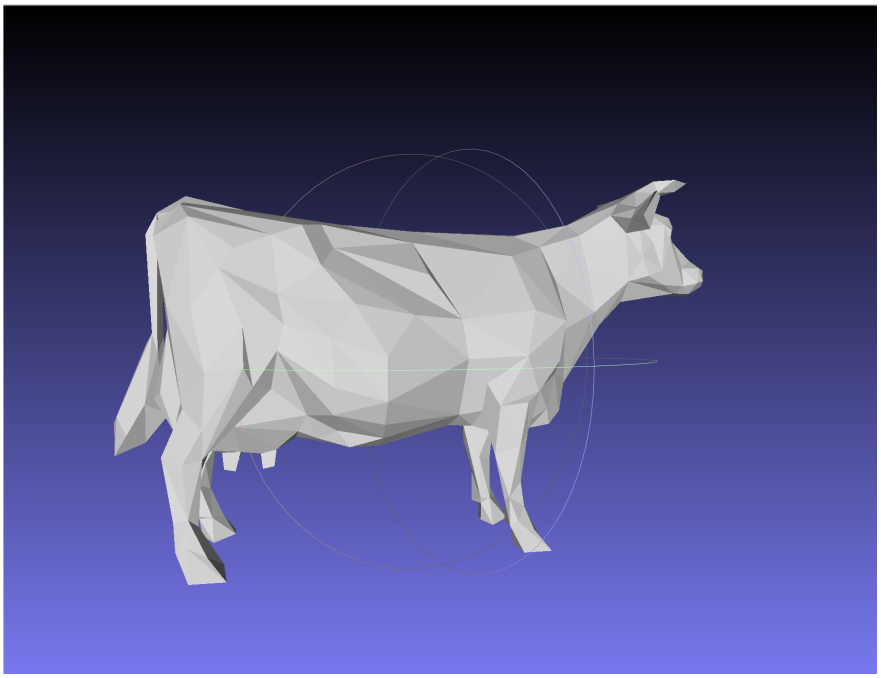

图 7: cow_simplified_5000.obj simplify 5000

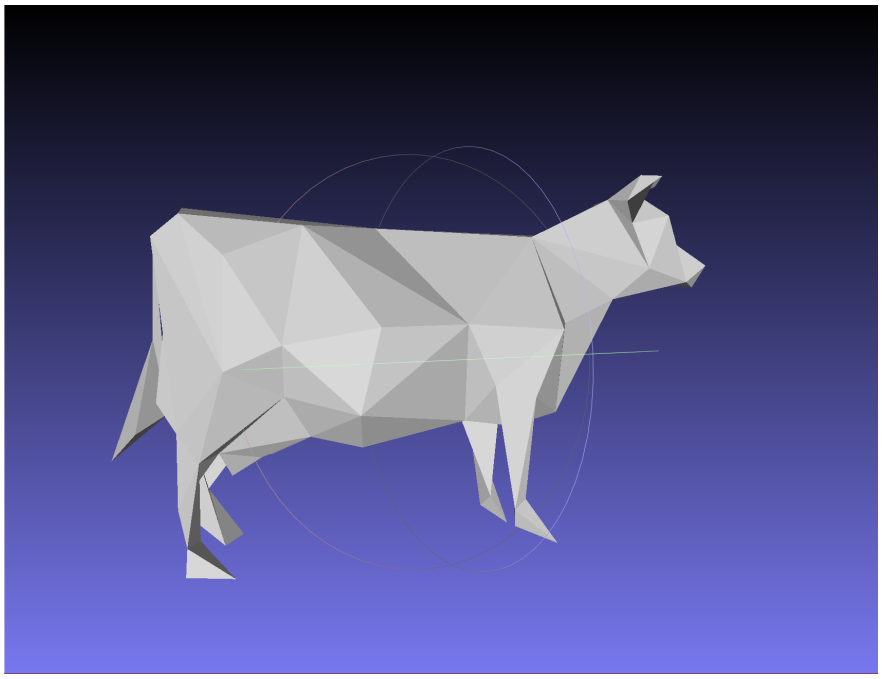

图 8: cow_simplified_5600.obj simplify 5600

There might be some problems in the cow.obj itself which meets the flaw of the contracting-based simplification algorithm, causing some flipping faces.

3 Validation

1. **Duplicate Faces Check**:

- **Purpose**: Detects if any two faces in the mesh are identical.
- **Action**: If a duplicate is found, an error message is printed, and the program asserts false.

2. **Invalid Vertex Indices Check**:

- **Purpose**: Ensures that all vertex indices referenced by faces are valid.
- **Action**: If an invalid index is found, an error message is printed, and the program asserts false.

3. **Unused Vertices Check**:

- **Purpose**: Checks whether every vertex is used in at least one face.
- **Action**: If a vertex is not used, a warning message is printed.

4. **Flipping Faces Check**:

- **Purpose**: Identifies faces that may be incorrectly oriented by checking edge usage.
- **Action**: If an edge appears more than twice, a warning message indicates a potential issue.

5. **Normals Consistency Check (Optional)**:

- **Purpose**: Calculates normals for each face to ensure they are correctly oriented.
- **Action**: This is optional and provides additional geometric validation.

6. **Connectivity Check**:

- **Purpose**: Ensures that all vertices in the mesh are connected through faces.
- **Action**: If any vertex is not reachable, a warning message is printed.

7. **Boundary Edge Check**:

- **Purpose**: Identifies edges that belong to only one face, indicating open boundaries.
- **Action**: If a boundary edge is detected, a warning message is printed.