2.5) Grid-based mesh simplification

Alim Dhanani, 101156584

December 13, 2023

Contents

1 Introduction

The project chosen is 2.5) Grid-based mesh simplification. Based on the algorithm proposed by Lindstrom [1, 2], the goal is assign a grid to a mesh, and simplify the mesh by clustering vertices together based on which cell they reside in. Simplifying meshes are useful for resource optimization, such as reducing storage utilization and rendering times [3].

The 3D models used in this project are all sourced from the GeomProc library [4], with a couple exceptions. The "Armadillo" was sourced from the Stanford Computer Graphics Laboratory, as was the "XYZ RGB Dragon" [5]. The source file for the XYZ RGB Dragon was found on the GitHub repository `alecjacobson/common-3d-test-models` [6].

1.1 Previous work

The methodology is based on Lindstrom's algorithm [2], including its general direction and use of hash maps ("hash table maps") as data structures. Lindstrom states,

> "[The] algorithm divides the model into cells from a uniform rectilinear grid, and replaces all vertices in a grid cell by a single representative vertex. When clustering vertices together, the majority of triangles degenerate into edges or points and can be discarded, thereby reducing the complexity of the model."[2]

Furthermore, Lindstrom describes his use of hash maps by giving the following steps,

- Given a triangle $t \in T_{in}$ from the original mesh, we fetch its vertex coordinates
- For each vertex v_{in} of t, we construct a hash key from the grid cell that the vertex falls in and do a hash table lookup
- This dynamic hash table maps grid cells, or clusters, to the vertices V_{out} in the simplified mesh [2]

However, the implementation in this project differs from Lindstrom's based on two factors: 1) quadrics are not used in this implementation as a heuristic to simplify vertices, and 2) the output representative vertex of a cell in this implementation is calculated using the weighed average of the cell's input vertices, each weighed by their Voronoi area in the original mesh.

2 Method

2.1 Overview of implementation

The implementation only requires two inputs: the input mesh (`mesh`) to be simplified, and the number of cells in each dimension (`num_cubes`). Note the assumption that the number of cells for all dimensions are equal; each cell has the same side length in each dimension, so for dimension $d = 3$ each cell is a cube.

The implementation normalizes the mesh so that it fits between -1 and 1 in all dimensions. Since the mesh is maximum of length 2 in any given dimension, the size of each cell will be $\frac{2}{num\_cubes}$. For example, if 10 cells are required then each cell will have length 0.2.

As stated in section 1.1, hash maps were used as data structures to store transitive information about the translation from original to simplified mesh. The first hash map (`vertex_id`) maps cell coordinates to an integer vertex id in the new, simplified mesh. The second hash map (`vertices_in_cell`) maps a cell to the array of all original-mesh vertices contained within that cell.

The overarching process' pseudo-code is given by Algorithm 1.

Algorithm 1: Pseudo-code for grid-based mesh simplification

```
Def SimplifyMesh(mesh, num_cubes):
    let d be the dimensionality of the mesh
    normalize mesh so that it fits from coordinates (−1, −1, −1) to (1, 1, 1)
    let n_old be the number of vertices in the original mesh
    grid ← initialize grid with num_cubes cells in each dimension
    let simplified-mesh be the output mesh
    let next-vertex-id ← 0
    let vertex_id be a hash-map
        key: hash of cell coordinates
        value: vertex id in new mesh (integer)
    let vertices_in_cell be a hash map
        key: hash of cell c_i's coordinates
        value: array of all v_j ∈ mesh such that v_j ∈ c_i
    foreach triangle t_in ∈ mesh.face do
        let cells be an empty array
        let hashes be an empty array
        foreach vertex index x_i ∈ t_in do
            v_i ← coordinates of vertex x_i in mesh
            c ← cell containing v_i
            push c to cells[]
            push hash(c) to hashes[]
        if not distinct(hashes) then        // if any vertices are in the same cell
            continue to next t_in                    // discard triangle completely
        foreach vertex index x_i ∈ t_in do
            let h_i be hashes[i]
            if h_i not in vertex_id then
                add h_i to vertex_id with value "next-vertex-id"
                create array in vertices-in-cell[h_i]
                increment next-vertex-id
            Add x_i to array in vertices-in-cell[h_i]
        t_out ← (vertex_id[h_1], vertex_id[h_2], vertex_id[h_3])
        Add t_out to simplified-mesh
    simplified-mesh vertices and v-normals ← calculateMergedVertices()    // Algorithm 2
    orient simplified-mesh's faces                                        // Algorithm 4
    return simplified-mesh
```

2.2 Calculating output vertex

As shown in Algorithm 2, to calculate each cell's output vertex positions and normals we first retrieve the mesh's Voronoi area information. Then, iterate through all the cells and find the weighed average of all the vertices contained in that cell (using the hash map `vertices_in_cell`), with the vertex's Voronoi area as the weight. This allows for more prominent vertices to be more heavily favoured in the representative vertex.

Algorithm 2: Calculating each cell's output vertex and vertex normal

```
1  Def CalculateMergedVertices(mesh, cells, vertices_in_cell):
2      let d be the dimensionality of the mesh
3      let n_new be the number of vertices in the simplified mesh
4      let new-vertices be a zero-filled 2D array with dimensions (n_new, d)
5      let new-vnormals be a zero-filled 2D array with dimensions (n_new, d)
6      fill new-vertices with 0
7      vareas ← get Voronoi areas of all vertices in mesh          // Algorithm 3
8      foreach c_i ∈ cells do
9          new-id ← retrieve the cell's new vertex id
10         cell-varea ← []
11         foreach vertex v_j in vertices_in_cell[hash(c_i)] do
12             cell-varea[j] ← varea[v_j]
13         total-varea = sum(cell-varea)
14         foreach vertex v_j in vertices_in_cell[c_i] do
15             weight ← cell-varea[j] ÷ total-varea
16             new-vertices[new-id] += weight * coordinates of v_j
17             new-vnormals[new-id] += weight * vnormal of v_j
18     return new-vertices, new-vnormals
```

2.3 Calculating Voronoi areas

To calculate the Voronoi areas for every vertex in a mesh, the problem was broken down: each vertices' total Voronoi area is the sum of its Voronoi area in each of its triangles. Thus, iterating over all triangles and adding each vertex's Voronoi area in that triangle will yield their total, as demonstrated in Algorithm 3. To calculate the Voronoi area for point A in a single triangle ABC, the following formula was used:

$$\mathrm{Vor}_A = \begin{cases} \frac{1}{2} area(\triangle ABC) & \text{if } (\cot(A) < 0) \\ \frac{1}{4} area(\triangle ABC) & \text{if } (\cot(B) < 0) \text{ or } (\cot(C) < 0) \\ \frac{1}{8}\left(||AC||^2 \cot(B) + ||AC||^2 \cot(C)\right) & \text{otherwise} \end{cases} \tag{1}$$

When a triangle is obtuse, its circumcenter will be outside of the triangle. To obtain a perfect tiling over the mesh surface, we can use a "mixed area": replace the circumcenter for obtuse triangles with the midpoint at the opposite edge (Figure 1). If D, E, F are the midpoints of AB, BC, AC respectively, then the medial triangle theorem states that drawing a triangle $\triangle DEF$ results in a partition of four triangles, each with exactly $1/4$ the area of $\triangle ABC$ [7], as shown in Figure 2. Thus, the "mixed-area" is divides the triangle into three parts where the largest partition is composed of two medial triangles and hence half the triangle area.

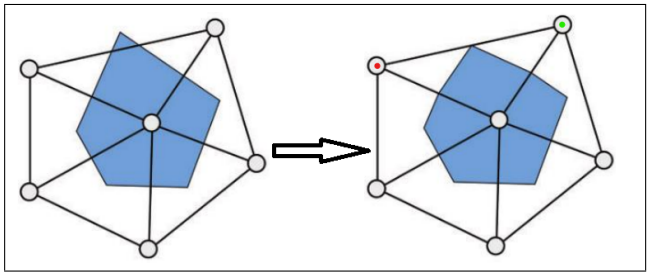

Figure 1: Using a mixed area: replace obtuse triangle's circumcenter with the midpoint at the opposite edge

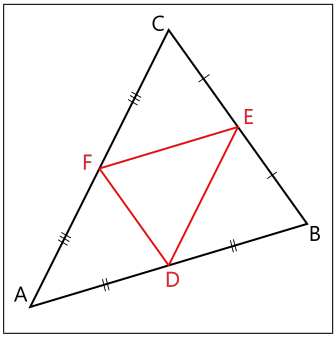

Figure 2: Medial triangle showing four equal-area sub-triangles of ABC [8]

We use $\cot(X) < 0$ to determine if angle X is obtuse. Thus, if A is obtuse ($cot(A) < 0$), return half the triangle area. If either angles B or C are obtuse, then the Voronoi area Vor_A is one quarter of the total triangle area.

Algorithm 3: Calculate Voronoi area of all vertices in a mesh

```
Def MeshVoronoi(mesh):
    let |V| be the number of vertices in the mesh
    initialize vareas as a zero-filled array with size |V|
    foreach f ∈ mesh.face do
        foreach i ∈ (0, 1, 2) do
            let A be the position of v_i ∈ f
            let B, C be the coordinates of the two other vertices in f
            vareas[v] ← vareas[v] + voronoiAngle(A, B, C)
            // calculate Voronoi angle using code from GeomProc[4]
    return vareas
```

2.4 Inverted triangles

Throughout the development of the project, we observed several "holes" in the simplified triangle meshes, particularly for complex models. Upon further inspection, these were not gaps in the mesh but rather **inverted triangles**, meaning the triangles were facing inwards (towards the interior of the mesh) rather than outwards. Most platforms who render 3D models either render these inverted triangles as black (such as MeshLab) implying no light reflects off it, or invisible altogether (Blender, Microsoft 3D Viewer). Thus, to ensure the visual appeal of the simplified meshes, a method was needed to identify inverted triangles, and flip them. Algorithm 4 describes the identification of the flipped faces after the simplification is complete, by calculating the dot product between the average vertex normals and the current face normal. If the dot product is less than 0, the face is facing in the opposite direction of the vertices and must be flipped. Algorithm 5 briefs the method used to invert a single triangle, by swapping the first two vertices.

Algorithm 4: Orient all triangle faces of a mesh so that they point outwards

```
Def OrientFaces(mesh, approx_vnormals):
    // approx_vnormals is calculated in Algorithm 2
    foreach f ∈ mesh.face do
        let v_1, v_2, v_3 be the vertices of f
        vertex-normals[1,2,3] ← approx_vnormals[v_1, v_2, v_3]
        approx-fnormal ← mean(vertex-normals)
        old-fnormal ← mesh.fnormal[f]
        dot ← old-fnormal dot-product approx-fnormal
        if dot < 0 then
            flipTriangle(f)                                  // Algorithm 5
```

Algorithm 5: Helper function for inverting a triangle's orientation

```
Def FlipTriangle(triangle):
    swap triangle[0] and triangle[1]
```

3 Results

The effect of varying the number of cubes on mesh simplification is akin to varying the resolution of the output simplified mesh. As seen in Figure 3, as the number of cells per dimension increases, more details are visible and it more closely resembles the original mesh. Intuitively, the low-resolution simplification results in the lowest number of vertices and faces (90% reduction). However, even with a fairly high-resolution simplification for the bunny ($n = 20$) the number of vertices ($|V|$) and triangles ($|F|$) are reduced by 60% (refer to Table 1). The time elapsed t only slightly varies when increasing the resolution, implying that the majority of the computation is classifying each vertex (which should be constant regardless of the number of cubes), and not re-building the new mesh.

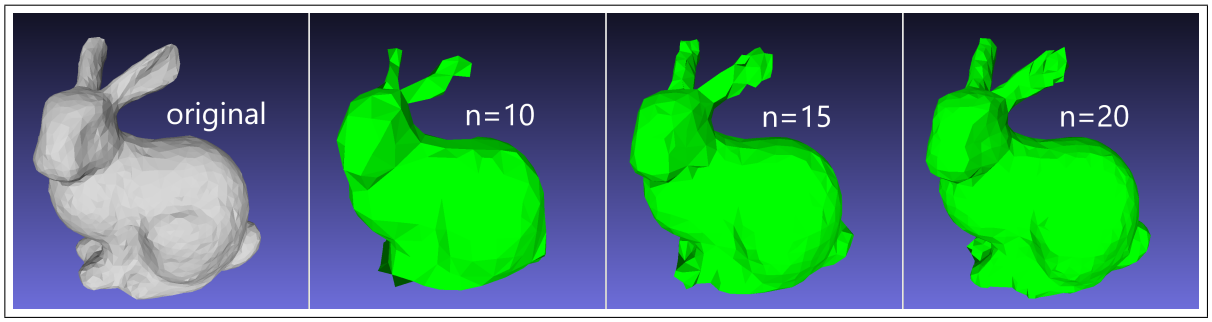

Figure 3: Varying the simplification resolution of the bunny

The implementation was also tested on more complex meshes with vertices and faces in the order of hundreds of thousands, such as the Armadillo figure. Using 50 cells per dimension, the simplified mesh has its number of vertices and triangles reduced by 97% (Figure 4). It is clear to see various topology changes, including the absence of key texture details such as the scales/ridges omnipresent on its surface and the bumps on its legs. Fine features such as the claws and ears

Number of cubes per dimension	Cube size	$\lvert V\rvert$	$\lvert F\rvert$	t
Original		**2,503**	**4,968**	
10	0.2	274	537	1.37s
15	$0.13\dot{3}$	621	1,225	1.42s
20	0.1	1,004	1,995	1.46s

Table 1: Effect of varying number of cubes (resolution) on bunny simplification

have lost their sharpness and appear choppy. Another key area of interest is the mouth, with clearly defined teeth and jaws turning into a simple-looking closed snout. However, the general shape of the Armadillo remains unaltered. The simplification of this more complex mesh required significantly more time, 96.52s compared to the bunny's 1.46s time elapsed.

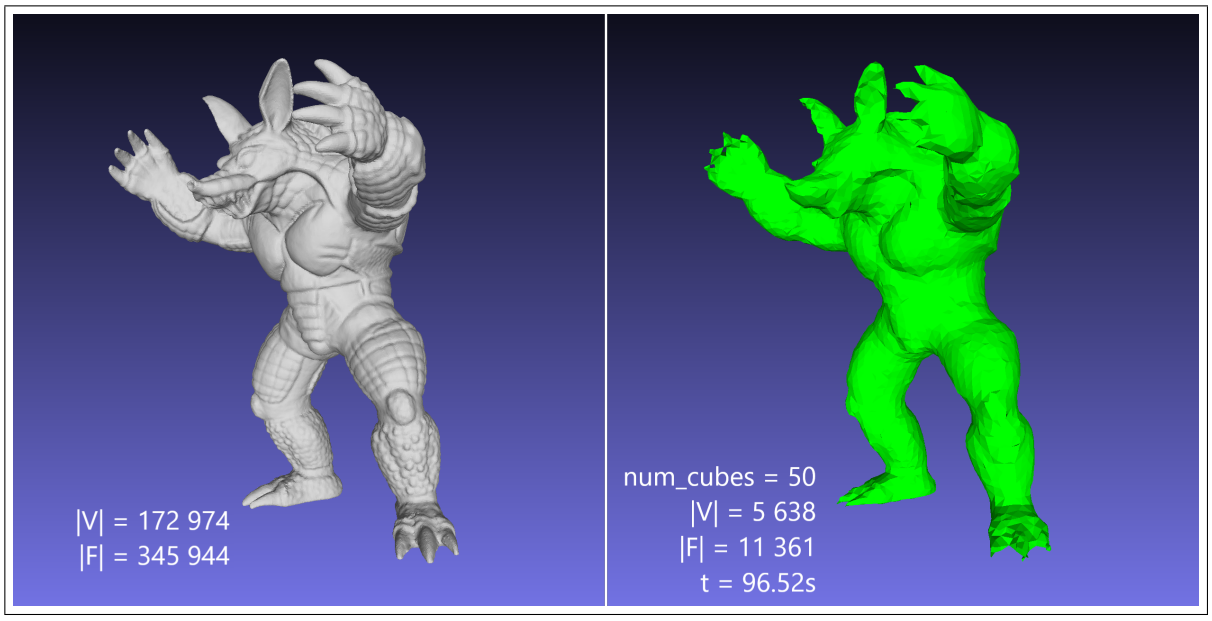

Figure 4: Simplifying the Armadillo mesh, resulting in a 97% reduction in vertices and triangles

3.1 Inverted triangles

As noted in section 2.4, an additional step of the project is to orient all the triangle faces so that they face outwards. Figure 5 demonstrates the difference between a figure without and with triangle inversion correction. Flipped triangles show as black, and some of which are encircled in red. With triangle inversion correction, 801 inverted triangles were identified and flipped. To add, the number of vertices and triangles remain constant as expected, since no triangles were formed or deleted. Also to note, the triangle inversion correction process only consumes an incremental amount of time, increasing the time elapsed from 67.37 seconds to 68.50.

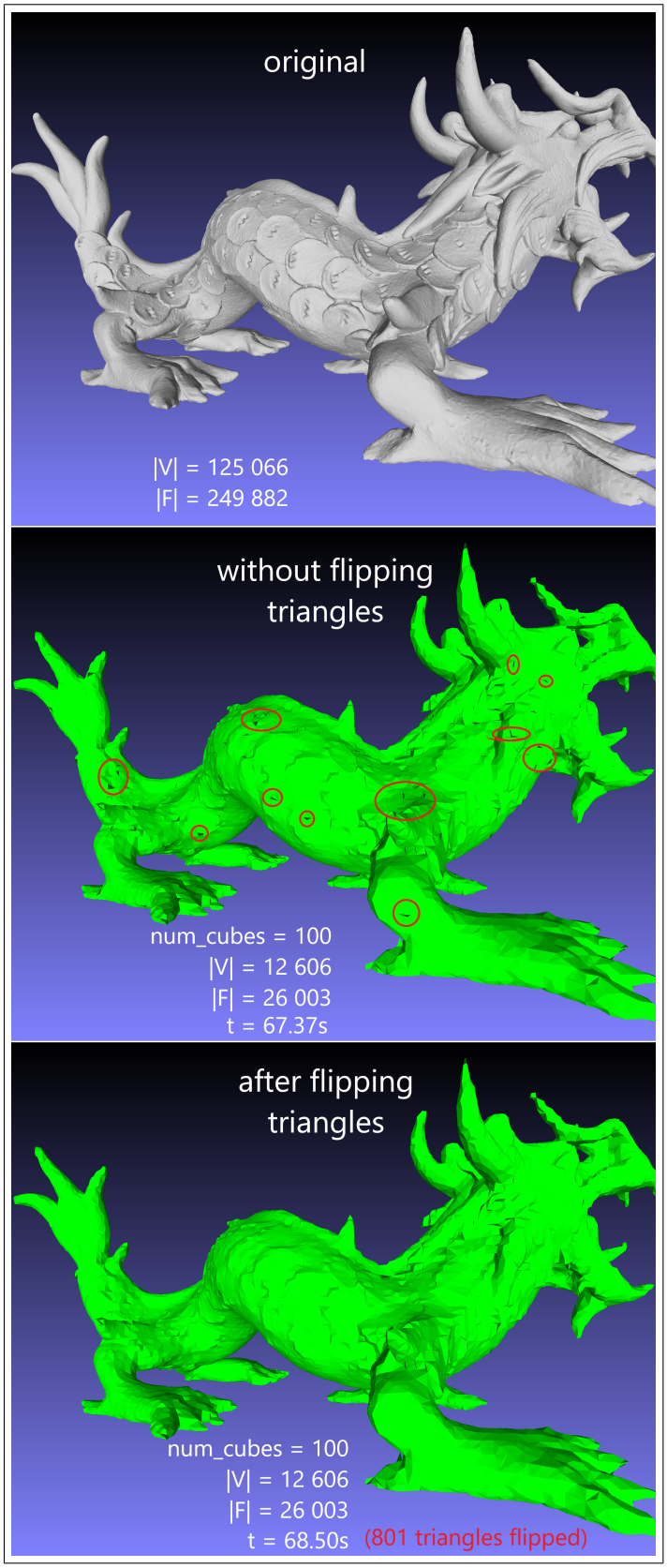

Figure 5: Simplifying the XYZ RGB Dragon object, without and with triangle inversion correction

4 Discussion

The method was fully implemented and functions moderately well as a means to simplify meshes. The simplified meshes lose finer details but retain overall shape.

The method achieves similar results to Lindstrom's 2000 paper [2], with the output meshes achieving similar levels of detail. Albeit surprisingly Lindstrom's method using quadrics accomplishes the simplification in a much shorter time span (16 seconds on a mesh with 870,000 triangles).

4.1 Limitations

Grid-based mesh simplification as a whole has one crucial flaw: manifold meshes can have their topology change and become non-manifold. For example, the camel object was simplified using 50 cells in each dimension (Figure 6). Although the simplification looks well-constructed, the "highlight non-manifold edges" feature in MeshLab identifies several non-manifold edges in red. Thus, this method is not advised where manifoldness of the simplified mesh is a requirement.

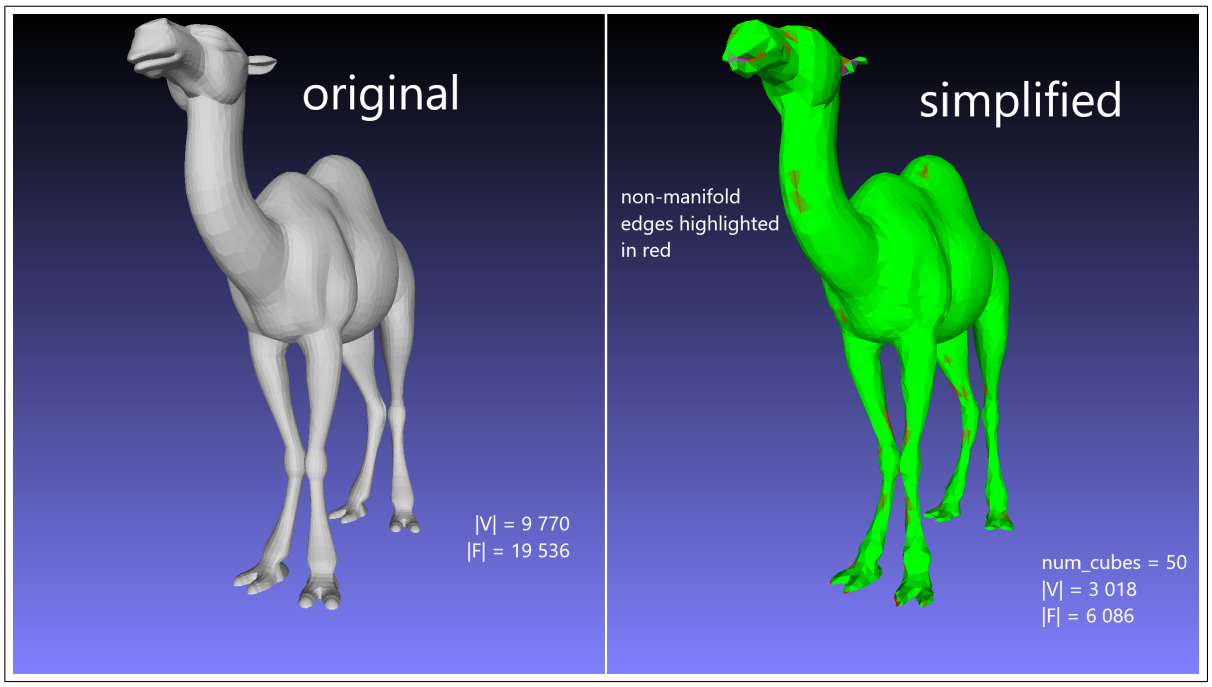

Figure 6: Camel simplification showing non-manifold edges in red

One of the key limitations of this implementation specifically is that it is not completed in "one pass", meaning that it requires a couple iterations over the mesh's data to complete the simplification process. This slows its overall performance. A future adjustment would be to update output vertices "on the fly" while analyzing the mesh, rather than iterating over the entire mesh,

then calculating output vertices.

Another aspect which could be changed is the initialization, that is, the creation of the grid and normalization of the mesh. Rather than scaling the input mesh to lie between -1 and 1, future work can be done to define the grid with respect to the mesh's minimum and maximum dimension. This avoids the process of mesh normalization, which can save computation time and resources.

References

[1] P. Lindstrom and G. Turk, "Fast and memory efficient polygonal simplification," in *Proceedings Visualization '98 (Cat. No.98CB36276)*, pp. 279–286, 1998.

[2] P. Lindstrom, "Out-of-core simplification of large polygonal models," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, (USA), p. 259–262, ACM Press/Addison-Wesley Publishing Co., 2000.

[3] M.-E. Algorri and F. Schmitt, "Mesh simplification," *Computer Graphics Forum*, vol. 15, no. 3, pp. 77–86, 1996.

[4] O. van Kaick, "Github - Ovankaic/GeomProc: Geometry Processing library." https://github.com/ovankaic/GeomProc, aug 22 2023. [Online; accessed 2023-09-13].

[5] S. C. G. Laboratory, "The stanford 3d scanning repository."

[6] alecjacobson, "alecjacobson/common-3d-test-models," Mar 2020.

[7] A. S. Posamentier and I. Lehmann, *The Secrets of Triangles: A Mathematical Journey*. Prometheus Books, aug 28 2012. [Online; accessed 2023-12-13].

[8] Braindrain0000, "File:medial triangle.svg." https://commons.wikimedia.org/w/index.php?curid=3699101, Mar 2008.