Implementation of Cage Generation with User Interaction in Cage Modeler

2024/25 Visual Computing Lab Report

TECHNISCHE UNIVERSITÄT DARMSTADT

Jinjoo Ha, Jiaqi Liu
March 31, 2025

Contents

1 Introduction **2**

2 Approach **3**
2.1 Voxelization . . . 3
2.2 Closing . . . 5
2.3 Remeshing . . . 6

3 Results **8**
3.1 Deformation with Generated Cage . . . 8
3.2 Runtime Analysis . . . 8
3.2.1 Stepwise Execution Time . . . 8
3.2.2 Runtime Comparison of Voxelization Algorithm . . . 9
3.3 Cage Results with Different Parameters . . . 9
3.3.1 Resolution . . . 9
3.3.2 Smoothing Factor . . . 10
3.3.3 Tri-Quad Mesh . . . 10

4 Conclusion **11**

5 References **12**

1 Introduction

Cage-based deformation is a versatile and fast approach for interactive manipulation of geometry. Given a cage, each vertex of the original mesh is assigned barycentric coordinates relative to the cage's vertices. These coordinates are computed once at the beginning and do not need to be recomputed afterward. When the cage's vertices are modified, the original mesh is deformed accordingly using the precomputed barycentric coordinates and the updated cage vertices. In CageModeler, many approaches have been implemented for cage-based deformation. However, there are still some issues related to cage generation, which is the purpose of this project.

Firstly, the existing method of cage generation is accomplished manually, which can be a time-consuming and cumbersome task. To deal with this issue, we went through A Survey on Cage-based Deformation of 3D Models by D. Ströter et al[2]. and compared numerous approaches about automatic cage generation. In this project, we examined the paper Bounding Proxies for Shape Approximation by Calderon and Boubekeur[1] and implemented a process of automatically generating a cage by simply inputting a model.

Secondly, cages generated from existing method are overly simplified and lack of tightness. To achieve more fine-grained deformation to the model, the generated cage is desired to possess the following properties: Bounding, Coarseness, Tightness, Geometry adaptiveness. In this project, we intend to realize a tighter and more geometrically adapted cage than the one generated by the existing method. The comparison of cages between our method and the existing method is shown in Figure 1. To verify that the quality of the generated cages meets the requirements of the cage deformation, we added the process of cage generation as a working module in the CageModeler application by D. Ströter et al.

Thirdly, user adaptiveness should also be considered in the process of cage generation. We added an interaction mechanism to enhance the user experience by giving the user more control over the features of the generated cage like voxel resolution, smoothness, face number and the type of mesh.

In addition, we have also made an attempt to generate Tri-Quad mesh cage. Tri-Quad mesh is a kind of mesh composed of triangular and quadrilateral units, the type of units in the different zones can be adjusted according to requirements. The quadrilateral part is usually used to maintain smooth surfaces, while the triangular part is used to deal with complex topological changes. Most of existing cage generation methods are based on triangular mesh and there are relatively less studies about automatic Tri-Quad cage generation. In this project, we added an option of Tri-Quad mesh in the process of cage generation for users to have targeted modification to the models.

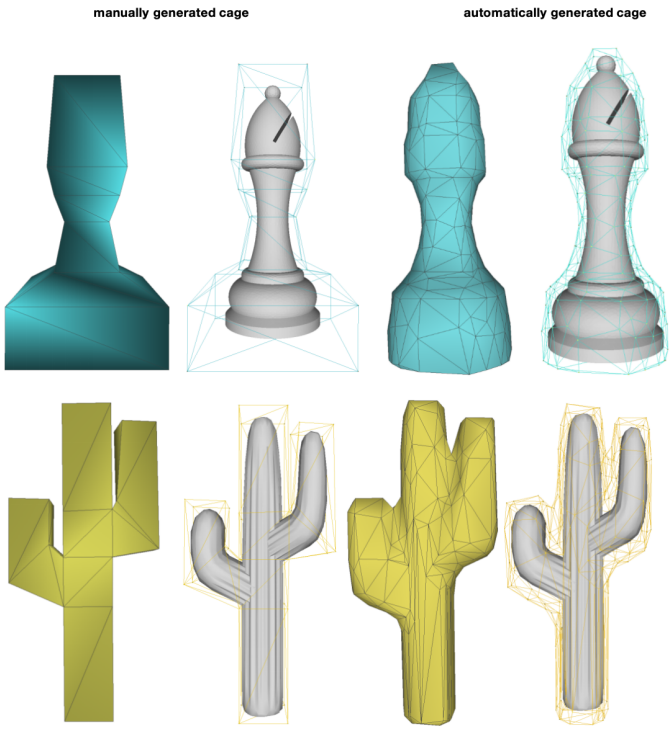


Figure 1: Comparison between Cages Generated from Existed Method and the Method in the Project

2 Approach

Cage generation in [1] is divided into three main steps: voxelization, morphological closing, and remeshing. Figure 2 shows the visualization of the intermediate steps and the output from the input mesh. Below, we examine each step in detail.

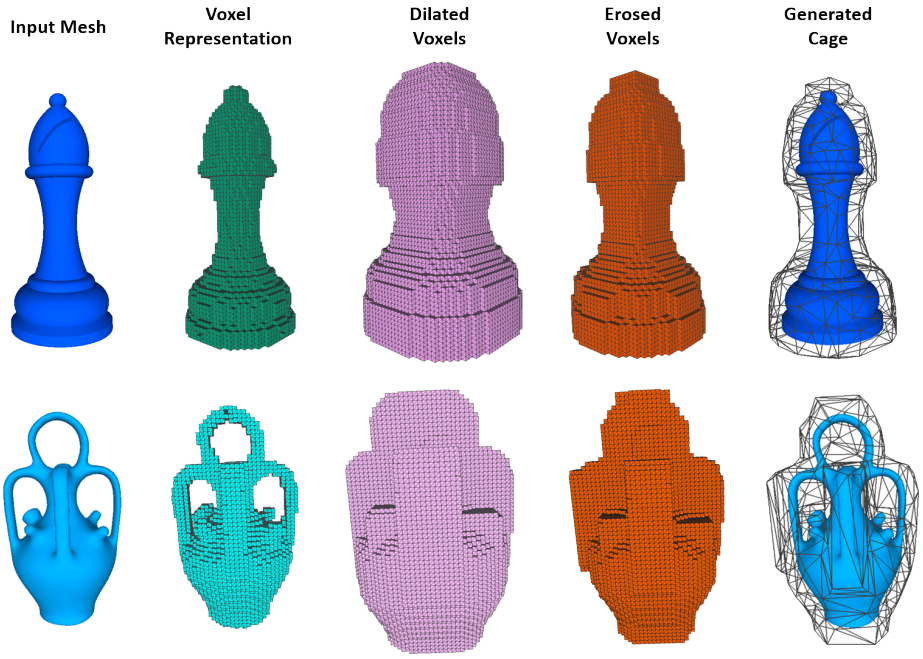


Figure 2: Overview of Cage Generation Steps (model: ChessBishop and Botijo)

2.1 Voxelization

Voxelization is the process of converting a surface mesh into a 3D binary grid, where each voxel indicates whether it is occupied or not. This step is the most time-consuming part of the entire cage generation process when using a naïve approach. To optimize performance, we implemented an accelerated voxelization algorithm utilizing OpenGL shaders, which was originally introduced in the implementation of [1] . This approach employs three types of shaders:

- **Vertex Shader**: Transforms vertices into normalized device coordinates (NDC), where the minimum position is (-1, -1, -1) and maximum position is (1, 1, 1). This operation is executed per vertex.

Listing 1: "vol_vox.vert"

```
#version 450

uniform vec3 bbox_min;
uniform float rescale;

in vec3 position;
in vec3 normal;

void main(void) {
    vec4 P = vec4 (rescale*(position.xyz - bbox_min) + vec3 (-1.f, -1.f, -1.f), 1.f);
    gl_Position = P;
}
```

- **Geometry Shader**: Generates and populates a data structure that is interpolated and passed to the fragment shader. This operation is executed per triangle.

Listing 2: "vol_vox.geo"

```
#version 450

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

out PerVertex
{
  vec4 vGridSpacePosition;
} gs_data_out;

out gl_PerVertex
{
  vec4 gl_Position;
};

void main() {
        vec3 p0 = gl_in[0].gl_Position.xyz;
        vec3 p1 = gl_in[1].gl_Position.xyz;
        vec3 p2 = gl_in[2].gl_Position.xyz;

        // Emit geometry
        gl_Position = vec4(p0, 1.0f);
        gs_data_out.vGridSpacePosition = vec4 (p0, 1.f);
        EmitVertex();
        gl_Position = vec4(p1, 1.0f);
        gs_data_out.vGridSpacePosition = vec4 (p1, 1.f);
        EmitVertex();
        gl_Position = vec4(p2, 1.0f);
        gs_data_out.vGridSpacePosition = vec4 (p2, 1.f);
        EmitVertex();
        EndPrimitive();
}
```

- **Fragment Shader**: Computes the voxel grid position by transforming the input vertex position from clip space to grid space. It then updates the voxel occupancy using imageAtomicXor, ensuring efficient atomic operations on a 3D texture image. Each value in the texture represents the occupancy of 32 voxels, compressed along the z-axis. The shader first marks the voxel corresponding to the fragment's position and then propagates occupancy updates down the z-axis, ensuring all lower voxels are correctly set.

Listing 3: "vol_vox_bit.frag"

```
#version 450

layout (binding = 0, r32ui) coherent uniform uimage3D vox_grid;

uniform uint res;

in PerVertex
{
  vec4 vGridSpacePosition;
} fs_data_in;

void main() {
        ivec3 final_grid_position;
        ivec3 uGridSize = ivec3 (res);
        vec3 clipSpacePosition = (fs_data_in.vGridSpacePosition.xyz + 1.0f) * 0.5f;
        final_grid_position = ivec3(clipSpacePosition * uGridSize);

        uint val_vox = 0xffffffff;
        int num_shifts = 31 - (final_grid_position.z % 32);
```

```
            val_vox = (val_vox >> num_shifts);
            final_grid_position.z = final_grid_position.z/32;
            imageAtomicXor (vox_grid, final_grid_position, val_vox);

            for (int i = final_grid_position.z - 1; i >= 0; i--) {
                    final_grid_position.z = i;
                    val_vox = 0xffffffff;
                    imageAtomicXor (vox_grid, final_grid_position, val_vox);
            }
    }
```

This three-step voxelization process is executed twice—once for volume voxelization and once for surface voxelization. The explanation above primarily focuses on volume voxelization.

Additionally, our implementation includes a fallback voxelization algorithm using CGAL, ensuring compatibility for platforms that do not support OpenGL 4.5.

2.2 Closing

Morphological closing is a fundamental operation in mathematical morphology used in image processing. It involves performing a dilation followed by an erosion, which helps smooth contours, fill small holes, and close narrow gaps in binary images. Closing is formally defined as:

$$G := E_a \circ D_b(G)$$

where E represents erosion, and D represents dilation. Erosion and dilation are respectively defined as:

$$D_a(G) = G \oplus B, \quad E_b(G) = G \ominus B$$

where $\oplus$ denotes the Minkowski sum, and $\ominus$ denotes the Minkowski subtraction. Morphological closing is effective for filling small holes or gaps while avoiding the filling of large gaps or the background, making it particularly suitable for cage generation. The paper introduces an efficient dilation and erosion algorithm that can be parallelized. This algorithm requires generating a mipmap of the voxel grid as a preprocessing step. Figure 3 and Figure 4 illustrates the algorithms.

Algorithem 1 Parallel dilatioin

```
Require: binary grid G
Require: a mipmap M generated from G

Function executeDilation(M,G):
    d_grid = mipmap[0]   // Base level voxel grid
    mipmap_depth = size of M

    num_voxels = {BASE_RESOLUTION, BASE_RESOLUTION, BASE_RESOLUTION}

    Parallel for x in range(BASE_RESOLUTION):
        for y in range(BASE_RESOLUTION):
            for z in range(BASE_RESOLUTION):
                node_stack = empty stack
                flat_idx = coords_to_voxel_idx(x, y, z, BASE_RESOLUTION)

                If d_grid[flat_idx] is occupied:
                    continue
                // Define structuring element (SE) based on resolution
                se = empty SE
                current_point = { level: 0, position: flat_idx }
                define_se(current_point, SE_SIZE * base_cellsize, se, false)

                // Start from the highest mipmap level
                node_stack.push({ level: mipmap_depth - 1, position: 0 })

                While node_stack is not empty AND d_grid[flat_idx] is not occupied:
                    top_node = node_stack.pop()
                    If top_node.level == 0:
                        d_grid[flat_idx] = true
                        Clear node_stack
                        break
                    Else:
                        subcells = find_subcells(top_node, M)
                        For each subcell in subcells:
                            subcell_val = mipmap[subcell.level][subcell.pos]
                            overlap = does_overlap(subcell, se)
                            If subcell_val is occupied AND overlap is true:
                                node_stack.push(subcell)
    Return d_grid
```

Figure 3: Algorithm for Dilation

Algorithem 2 Parallel Erosion

```
Require: binary grid G
equire: binary grid D          -> Dilation result D
Require: a mipmap M'           ->generated from D', the contour of D

Function execute_erosion(M', G, D):
    mipmap_depth = size of M'
    E=D
    Parallel for x in range(BASE_RESOLUTION):
        for y in range(BASE_RESOLUTION):
            for z in range(BASE_RESOLUTION):
                flat_idx = coords_to_voxel_idx(x, y, z, BASE_RESOLUTION)

                If E[flat_idx] is empty OR G[flat_idx] is occupied:
                    continue

  p_bbox = calc_voxel_bbox(flat_idx, BASE_RESOLUTION, global_min_point, base_cellsize)
                node_stack = empty stack
                node_stack.push({ level: mipmap_depth - 1, position: 0 })

                While node_stack is not empty AND E[flat_idx] is occupied:
                    top_node = node_stack.pop()

                    If top_node.level == 0:
                        E[flat_idx] = empty
                        Clear node_stack
                        break
                    Else:
                        subcells = find_subcells(top_node, M')
                        For each subcell in subcells:
                            subcell_val = M'[subcell.level][subcell.pos]
                            overlap = does_overlap_erode(subcell, p_bbox)
                            If subcell_val is occupied AND overlap is true:
                                node_stack.push(subcell)
```

Figure 4: Algorithm for Erosion

2.3 Remeshing

Remeshing in the cage generation is the process of transforming the polyhedron composed of voxel blocks obtained from the closing phase into a closed mesh. It includes two steps: surface extraction and mesh decimation.

- Surface Extraction

  The main idea in this step is to identify the exposed faces of each voxel block that makes up a polyhedron and extract them as triangular faces to form the surface of the polyhedron. An exposed face is defined as a voxel that has no neighboring voxel in the direction of that face. Figure 5 shows the process of surface extraction.

- Mesh Decimation

  The purpose of this step is to simplify the extracted surface mesh within a specific error margin, reducing the number of faces in the mesh to obtain a fully wrapped cage that fits the original model as closely as possible. We applied HC(Humphrey's Classes)Laplasian Smoothing to the surface mesh first to ensure the tightness without excessive shrinkage of the cage. To reduce the number of mesh face, we used QEM(Quadric Error Metrics)-based edge collapse in vcg library for mesh simplification. The main idea of QEM is : for every vertex v, define an error matrix Q, which measures the error from the vertex to all neighboring faces:

  $$Q = \sum (i = 1, n) K_i$$

  K_i is the error matrix of every face i, which is decided by the face normal vector (a, b, c, d)

  $$K_i = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

  for a vertex $v = (a, b, c, 1)$, its error is calculated from

  $$v^\top Q v$$

  in the edge collapse phrase, each time choose the edge determined by the two vertices with the smallest sum of errors is for collapsing, the position of the new vertex v' is determined by:

  $$v' = \arg\min_v v^\top (Q_1 + Q_2) v$$

Algorithem 3 voxelized polyhedron surface extraction

```
Require: binary grid G
Require: voxel_strides[3]   -> voxel stride in 3 directions
Require: global original point

Define NEIGHBORS = [
    (−1, 0, 0), (1, 0, 0),
    (0, −1, 0), (0, 1, 0),
    (0, 0, −1), (0, 0, 1)
]

Define FACES = [
    [0, 3, 6, 2],   // −X
    [1, 4, 7, 5],   // +X
    [1, 5, 3, 0],   // −Y
    [2, 6, 7, 4],   // +Y
    [1, 0, 2, 4],   // −Z
    [3, 5, 7, 6]    // +Z
]

Function compute_vertices(voxel origin, voxel_strides):
      RETURN 6 vertices of the voxel

Function add_faces(mesh, vertices, exposed_faces):
    FOR i IN RANGE(6):
        IF exposed_faces[i]:
            mesh.add_triangle(vertices[FACES[i][0]],vertices[FACES[i][1]], vertices[FACES[i][2]])
            mesh.add_triangle(vertices[FACES[i][2]], vertices[FACES[i][3]], vertices[FACES[i][0]])


Function extract_surface(grid, voxel_strides, global_origin, outputfile):
    INITIALIZE mesh
    FOR (x, y, z) IN OCCUPIED_CELLS(grid):
        vertices = compute_vertices(origin + x * voxel_strides[0] + y * voxel_strides[1] + z *
voxel_strides[2], voxel_strides)
        exposed_faces = [grid[x+dx][y+dy][z+dz] IS EMPTY FOR (dx, dy, dz) IN NEIGHBORS]
        add_faces(mesh, vertices, exposed_faces)
   SAVE mesh TO outputfile
```

Figure 5: Algorithm for Surface Extraction

- Tri-Quad Mesh Generation

  In the Project Settings Panel of CageModeler, we provide users with the option for generating cage as a Tri-Quad mesh. The process of Tri-Quad Mesh building is based on the triangular mesh generated in the previous phase. The main idea is as follows: for each triangular face, the angle between it and all its neighboring faces is calculated. The closest co-planar neighbor is then identified, and the two faces are merged into a quadrilateral. This process is iterated until no neighboring faces can be merged, and the remaining faces will stay triangular. The pseudo-code for this process is shown in Figure 6

Algorithem 4 convert triangular mesh into Tri-Quad mesh

```
Require: Triangular mesh T
Require:   face angle threshold θ
Function compute_angle_between_faces(T, face1, face2):
      RETURN normal_angle(face1,face2)

Function find_best_merge(T, face f, angle_threshold, merged_faces):
    min_angle = angle_threshold   # Initialize with the max allowable angle
    best_halfedge = NULL
    best_neighbor = NULL
    FOR each edge h of f:
        IF neighbor(f,h) is NULL OR already merged THEN continue
       angle = compute_angle_between_faces(T, f, neighbor(f,h))
        IF angle < min_angle:
            min_angle = angle
            best_halfedge = h
            best_neighbor = neighbor
    IF best_neighbor is NOT NULL:
        Merge face and best_neighbor along best_halfedge
        Mark face and best_neighbor as merged
        RETURN true
    ELSE:
        RETURN false

Function convert_to_tri_quad_mesh(T, θ = 10.0):
    merged_faces = empty set   # Keep track of merged faces
    faces_list = All triangle faces in the mesh

    FOR face in faces_list:
        IF face is already merged THEN continue
        find_best_merge(T, face, θ, merged_faces)
```

Figure 6: Algorithm for Tri-Quad Cage Generation

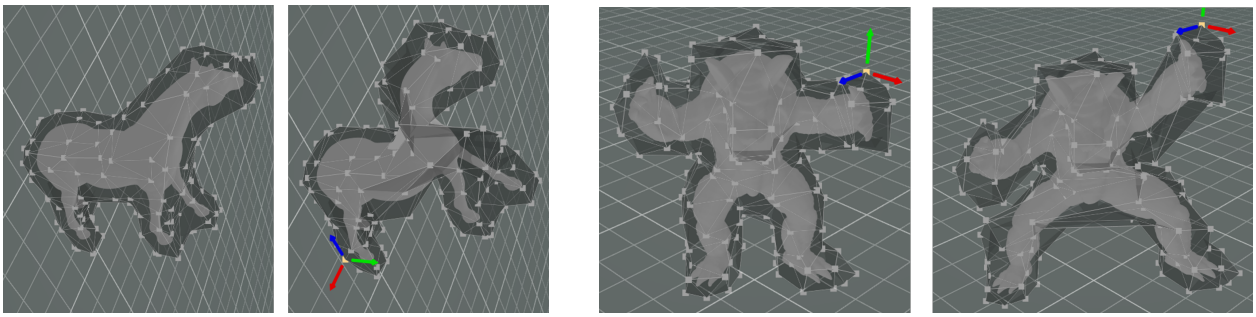
In the process of remeshing mentioned above, there are some factors that can cause non-closure, self-intersection, and face-flip in the generated mesh, which can be fixed by some functions of the CGAL library.

3 Results

In this section, we present the results of our experiments and analyses related to the mesh deformation, runtime performance, and the impact of different parameters on the generation and quality of the cage.

3.1 Deformation with Generated Cage

Figure 7 presents the outcomes of the mesh deformation using the generated cage method within the CageModeler UI.



(a) Cage Deformation of Horse

(b) Cage Deformation of Armadilloman

Figure 7: Mesh Deformation with our generated cage

3.2 Runtime Analysis

We conducted a runtime analysis of the main steps of the pipeline and of the different implementations of voxelization. The elapsed times were measured up to 10 times and averaged. Some minor intermediate steps, such as mipmap generation and contour extraction, are not included in this anlysis. We conducted these experiments on a machine with the following specifications:

- **CPU**: Intel Core i7-8550U @ 1.80GHz (4 cores)
- **GPU**: Intel(R) UHD Graphics 620 (7.9GB)
- **RAM**: 16GB
- **Operating System**: Windows 10
- **Software**: Visual Studio 2022

3.2.1 Stepwise Execution Time

We observed that the performance of the accelerated voxelization depends on the number of faces, rather than the voxel grid resolution. Other steps, such as dilation, erosion, and remeshing, are heavily affected by the voxel resolution but not by the number of faces.

Model	Resolution	Voxelization	Dilation	Erosion	Remeshing	Total
ChessBishop (# of faces: 16,310)	32	0.49	0.22	0.05	0.20	**0.97**
	64	0.42	2.84	0.87	0.98	**5.24**
	128	0.40	76.06	20.40	5.26	**105.46**
Bench (# of faces: 130,812)	32	1.37	0.26	0.05	0.14	**1.83**
	64	1.35	2.67	0.76	0.68	**5.56**
	128	1.41	67.24	18.70	3.34	**94.25**

Figure 8: Breakdown of computation time for different stages of the algorithm (in seconds).

3.2.2 Runtime Comparison of Voxelization Algorithm

From Figure 9, we can see that the accelerated voxelization algorithm provides significant benefits, particularly at higher resolutions.

Model	Resolution	CGAL w/ OpenMP	OpenGL Shader
ChessBishop (# of faces: 16,310)	32	0.87	0.39
	64	3.61	0.42
	128	31.94	0.42
Bench (# of faces: 130,812)	32	8.78	1.46
	64	9.88	1.65
	128	15.61	1.40

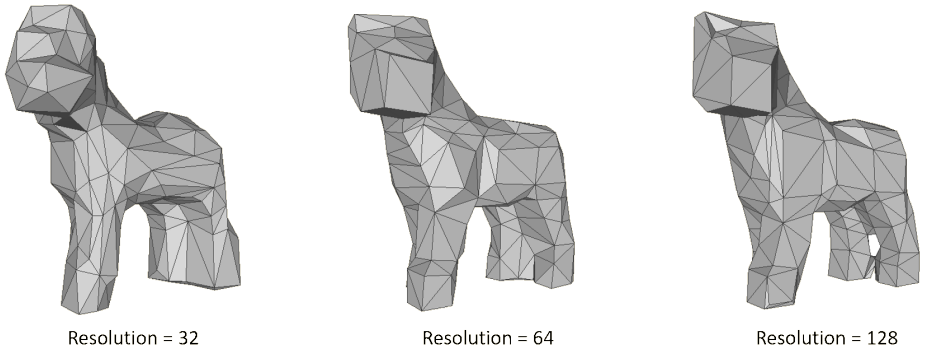
Figure 9: Comparison of voxelization algorithm runtime using different implementations: (1) CGAL with OpenMP parallelization, and (2) OpenGL shader acceleration (in seconds).

3.3 Cage Results with Different Parameters

We compared the results by varying several parameters that influence the resulting cage: the resolution of the voxel grid, the smoothing factor, and the choice of Tri-Quad mesh. For each test, we changed only the parameter of interest while keeping the other parameters fixed.

3.3.1 Resolution

In Figure 10, the cage generation results with different voxel reolutions(32, 64, 128) are compared.



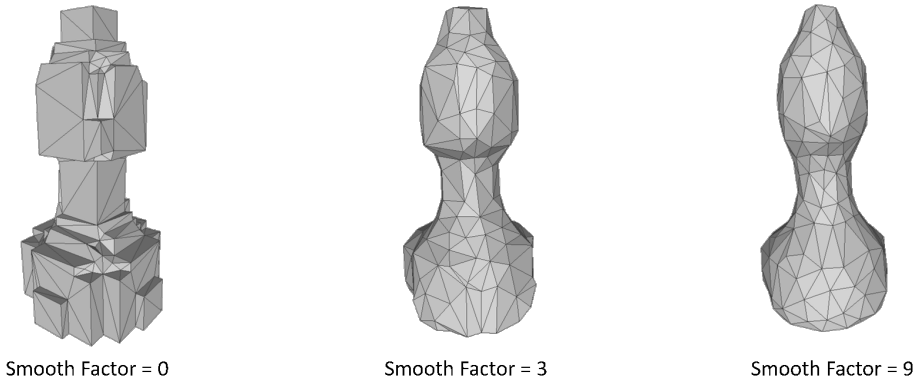
(a) Horse Cages with Different Resolutions

(b) Armadillo Cages with Different Resolutions

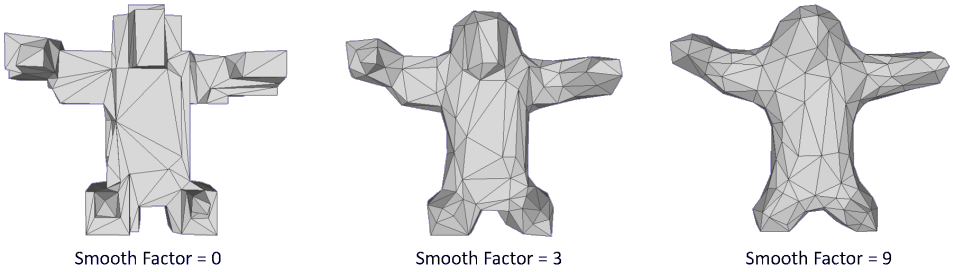
Figure 10: Comparison of Results with Different Voxel Resolutions

3.3.2 Smoothing Factor

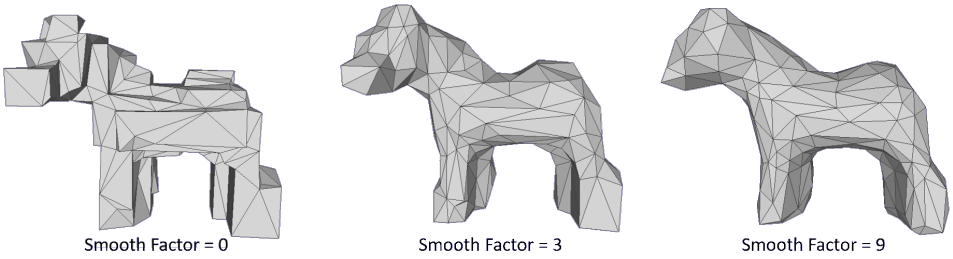
We can observe how the cage changes depending on the smoothing factor in Figure 11.



(a) ChessBishop Cages with Different Smoothing Factors



(b) Orge Cages with Different Smoothing Factors



(c) Horse Cages with Different Smoothing Factors

Figure 11: Comparison of Results with Different Smoothing Factors

3.3.3 Tri-Quad Mesh

Figure 12 and Figure 13 show the Tri-Quad mesh generated from triangular mesh and the deformation conducted by Tri-Quad cage, respectively.

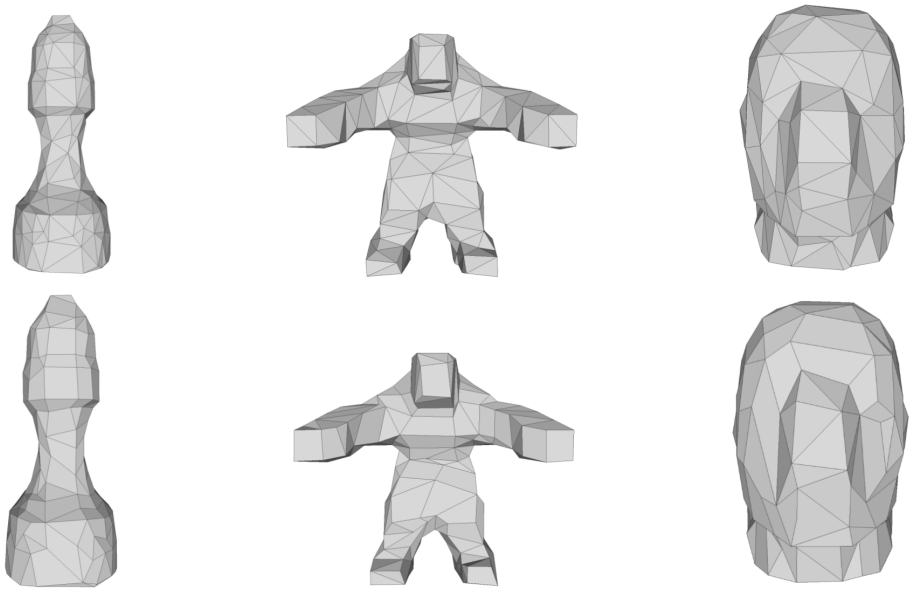
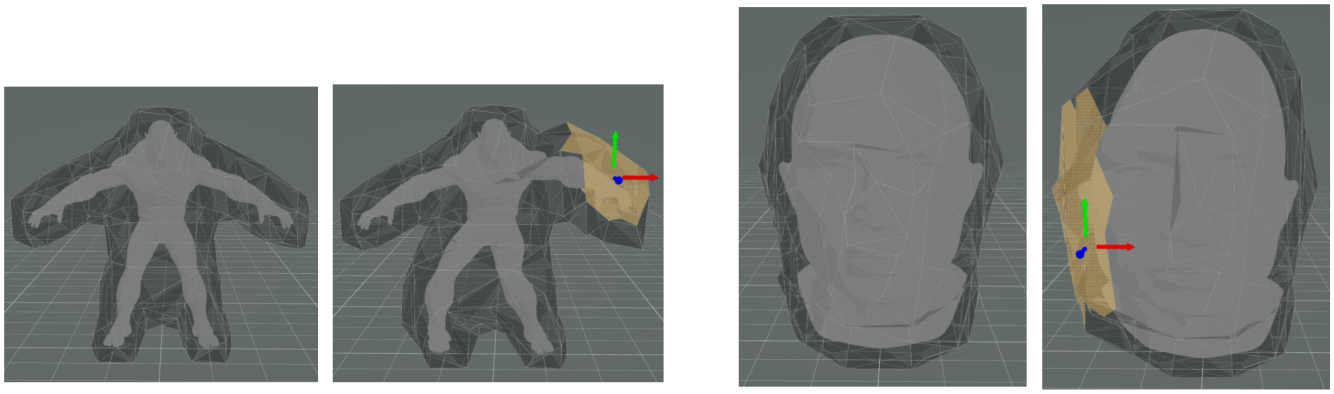


Figure 12: Triangular mesh(top) and corresponding Tri-Quad mesh(bottom) of model: chessBishop, ogre, manHead



(a) Tri-Quad Cage Deformation of Beast

(b) Tri-Quad Cage Deformation of Man Head

Figure 13: Tri-Quad Mesh Deformation with our generated cage

4 Conclusion

We have implemented automatic cage generation and demonstrated its usability and effectiveness for mesh deformation. Additionally, we enabled user interaction by giving users the control over voxel resolution, smoothing factor and coarseness of the cage faces.

We also accelerated the voxelization step, which was highly time-consuming in our initial implementation, by utilizing OpenGL Shader algorithm from [1]. As a result, it enabled the cage generation process at higher resolutions.

In addition, we have completed the automatic generation of the Tri-Quad mesh cage, which can also accomplish the deformation. We also found that to make the generated Tri-Quad cage actually show its geometric feature in the CageModeler, user need to use Quad Mean Value Coordinates (QMVC) or Green Coordinates for Quad cages (QGC).

Although our method allows users to set parameters for smoothness and the number of faces, when these parameters reach specific values, the resulting cage may intersect with the model. However, the deformation still works in the release version. We found that these thresholds vary across different models. Therefore, determining the relationship between parameter thresholds and model features, and developing algorithms to address this issue, should be the next step of this project.

Additionally, optimizing the dilation and erosion steps could also be considered for future work. Although these steps are currently parallelized using OpenMP, GPU acceleration could significantly reduce the runtime.

5 References

References

[1] Stéphane Calderon and Tamy Boubekeur. Bounding proxies for shape approximation. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(5), july 2017.

[2] D. Ströter, J. M. Thiery, K. Hormann, J. Chen, Q. Chang, S. Besler, J. S. Mueller-Roemer, T. Boubekeur, A. Stork, and D. W. Fellner. A survey on cage-based deformation of 3d models. *Computer Graphics Forum*, 43(2):e15060, 2024.