

GSDeformer: Direct, Real-time and Extensible Cage-based Deformation for 3D Gaussian Splatting

Jiajun Huang, Shuolin Xu, Hongchuan Yu, and Tong-Yee Lee

Abstract—We present GSDeformer, a method that enables cage-based deformation on 3D Gaussian Splatting (3DGS). Our approach bridges cage-based deformation and 3DGS by using a proxy point-cloud representation. This point cloud is generated from 3D Gaussians, and deformations applied to the point cloud are translated into transformations on the 3D Gaussians. To handle potential bending caused by deformation, we incorporate a splitting process to approximate it. Our method does not modify or extend the core architecture of 3D Gaussian Splatting, making it compatible with any trained vanilla 3DGS or its variants. Additionally, we automate cage construction for 3DGS and its variants using a render-and-reconstruct approach. Experiments demonstrate that GSDeformer delivers superior deformation results compared to existing methods, is robust under extreme deformations, requires no retraining for editing, runs in real-time, and can be extended to other 3DGS variants. **Project Page:** <https://jhuangbu.github.io/gsdeformer/>

Index Terms—3D Gaussian Splatting, Cage-based Deformation, 3D Representation Editing, Animation

I. INTRODUCTION

3D Gaussian Splatting (3DGS) [1] is a novel and efficient approach for reconstructing and representing 3D scenes. Due to its ability to capture real-world objects and environments in impressive quality, it holds significant potential for downstream applications such as animation, virtual reality, and augmented reality. To make 3DGS practical for these applications, it is crucial to enable users to freely edit the captured scenes for privacy or creative purposes.

Current methods do not achieve direct, real-time, and extensible manipulation of 3D Gaussian Splatting (3DGS). Techniques like DeformingNeRF [2], CageNeRF [3], and NeRF-Shop [4], which enable cage-based deformation on Neural Radiance Fields (NeRF), work by deforming sample points during the volumetric rendering process. However, since 3DGS does not use volumetric rendering, these methods cannot be easily adapted to it, restricting them to the more expensive-to-train and lower-quality NeRF-based representations.

While existing methods for editable 3DGS, such as SuGaR [5] and Gaussian Frosting [6], achieve high-fidelity deforma-

tion by tightly binding Gaussians to a high-resolution proxy mesh, they require significant modifications to the 3DGS representation or impose additional requirements on the training data, leading to the need for retraining. This limitation prevents these methods from directly editing existing 3DGS captures without extensive retraining, also making them harder to integrate with other 3DGS-derived scene representations. Similarly, methods like SC-GS [7] and VR-GS [8] that allow manual editing rely on control structures such as tetrahedral grids or control point clouds, which are less convenient for integration with existing animation software and pipelines.

To address these challenges, we propose GSDeformer, a method that enables cage-based deformation on trained vanilla 3DGS models and their variants. Our approach operates directly on trained 3DGS without requiring extensive retraining, performs deformation in real time, and can be easily extended and integrated with other methods that enhance or build upon 3DGS.

Our approach leverages Cage-Based Deformation (CBD), which uses a coarse mesh (cage) to control deformations of the finer geometry within it. To deform the Gaussian distributions that make up the 3DGS representation, we generate a proxy point cloud from the Gaussians and apply CBD to deform the point cloud. The deformed proxy points then drive the transformations applied to the Gaussians. To handle potential bending caused by deformation, we introduce a splitting process for the relevant Gaussians.

This direct deformation approach enables editing of any trained vanilla 3DGS model without requiring architectural modifications or retraining. It also ensures compatibility with other 3DGS variants. Additionally, by using standard triangular cage meshes to control deformation, our method seamlessly integrates with existing animation software and pipelines.

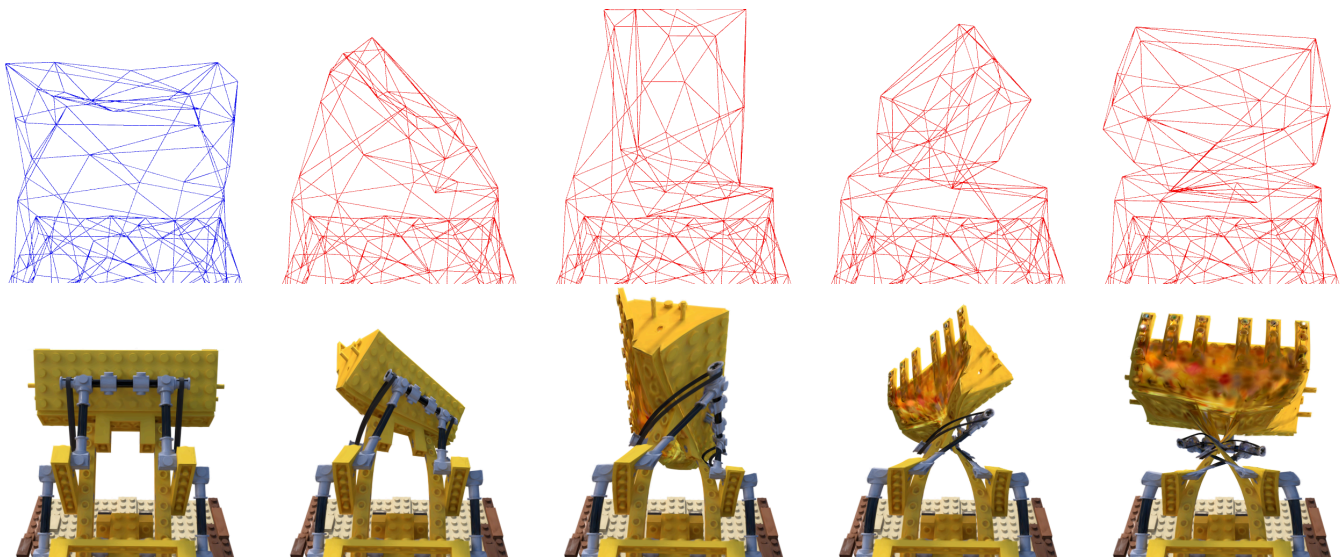
Cages for deformation can be created either manually or automatically from 3DGS using our automated algorithm. Our automatic cage-building algorithm supports not only 3DGS but also its variants, such as 2DGS [9], through a render-reconstruct-simplify approach. We also introduce a two-stage optimization design into our final mesh simplification process. This process is designed to produce a high-quality cage by both ensuring it better envelops the object to be deformed and improving the quality of the resulting MVC coordinates, which helps reduce artifacts in downstream deformation.

We evaluate the effectiveness of our method on object datasets. The results demonstrate that our deformation algorithm delivers superior quality compared to existing methods, particularly under extreme deformations. Among methods that enable explicit control, our control structure is the easiest to

This work was supported by HORIZON EUROPE Marie Skłodowska-Curie Actions, Grant 101130271, EPSRC Centre for Doctoral Training in Digital Entertainment, Grant EP/L016540/1, and the National Science and Technology Council, Taiwan, Grant 114-2221-E-006-114-MY3. (Corresponding authors: Hongchuan Yu, Tong-Yee Lee.)

Jiajun Huang, Shuolin Xu, and Hongchuan Yu are with the National Centre for Computer Animation, Bournemouth University, Poole BH12 5BB, U.K. (e-mail: jhuang@bournemouth.ac.uk; sxu@bournemouth.ac.uk; hyu@bournemouth.ac.uk).

Tong-Yee Lee is with the Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan 70101, Taiwan, China. (e-mail: tonylee@mail.ncku.edu.tw).



Src Cage + Dst Cage + 3DGS → Deformed 3DGS
No Retrain ✓ Real-time ✓ Extensible ✓

integrate with existing software. Furthermore, our algorithm achieves real-time performance ($\sim 60FPS$), renders deformed representations at high speeds ($> 200FPS$), and can be extended to other 3DGS variants.

In summary, our contributions include:

- We propose GSDeformer, a method achieving real-time cage-based deformation on any trained 3D Gaussian Splatting(3DGS) model without re-training or altering its core architecture.
- We also propose a robust automatic cage-generation algorithm that works on 3DGS as well as its variants due to its render-reconstruct-simplify approach. It ensures the creation of a high-quality cage by encouraging it to envelop the object better and produce stable Mean Value Coordinates through a two-stage process.
- We conduct extensive experiments to demonstrate our method’s ease of integration with other work extending 3DGS, as well as showing our method’s superior quality and ease-of-control against existing methods under normal and extreme scenarios.

II. RELATED WORK

A. Deformation of 3D Gaussian Splatting Scenes

Many methods have been proposed to edit 3D Gaussian Splatting (3DGS) models. These range from high-level, textual-prompt-based editing methods such as GaussianEditor [10], VcEdit [11], and GaussCtrl [12], to lower-level, more explicit editing methods that directly manipulate the scene’s geometry.

To enable explicit editing, one effective approach is binding the Gaussian distributions in 3DGS to the surface of a dense proxy mesh surface. Deforming the 3DGS scene can then be

achieved by deforming this mesh. This strategy allows for fine-grained, high-fidelity control over the geometry, as the high-resolution mesh provides a well-defined surface structure that can preserve intricate details during deformation. SuGaR [5] pioneered this approach by extracting a mesh from a 3DGS scene, and GaussianFrosting [6] builds on it by proposing a more flexible way to bind distributions to the mesh. Other methods like GaMeS [13] and Gao et al. [14] start with a provided initial mesh, while Mani-GS [15] first trains a 3DGS or NeuS [16] model to create one. However, the trade-off for this high-fidelity control is that the binding process modifies the core 3DGS architecture to handle deformation, requiring costly retraining. This limitation makes it challenging to use these methods ([5], [6], [13]–[15]) for modifying existing, pre-trained 3DGS scenes or extending them to new variants of the standard 3DGS. In contrast, our method prioritizes flexibility and compatibility by directly operating on the Gaussian primitives, eliminating the need for retraining or architectural changes and making it easier to integrate with variants of 3DGS.

In another direction, many works have explored physics-based simulation to manipulate 3DGS. PhysGaussian [17] uses the Material Point Method (MPM) [18] to model object deformation from touch and push interactions on 3DGS, and Gaussian Splashing [19] integrates position-based dynamics (PBD) [20] with 3DGS for simulations. Similarly, VR-GS [8] employs a dense tetrahedral grid to enable real-time physical interaction. While the grid-based deformation mechanism bears some resemblance to cage-based approaches, its primary goal is simulating physical phenomena rather than providing the direct, manual manipulation central to our method. Overall, while these methods excel at replicating natural physics, our approach focuses on direct, user-controlled artistic deforma-

tion and manipulation.

Other work has explored direct manipulation of vanilla 3DGS models without requiring a dense proxy mesh. SC-GS [7] enables Gaussian deformation by mapping control point transformations to Gaussians, but it requires video data for learning these mappings, limiting its applicability. D-MiSo [21], similar to SC-GS, focuses on editing dynamic 3DGS scenes rather than static ones. Recent work ARAP-GS [22] adapts the As-Rigid-As-Possible deformation approach and resolves spike-like artifacts by aligning deformed Gaussians with their corresponding radiance fields. However, this method can be less effective in handling extreme deformations, such as the Gaussian highlighted in Figure 15, which shows the bending of a single Gaussian. Our observation is that splitting bent Gaussian ellipsoids into multiple smaller ones is essential. Our method addresses this limitation by introducing a novel splitting mechanism, which divides bent Gaussian ellipsoids into multiple smaller ones to better represent the deformed shape.

B. Cage-Based Deformation

Cage-based deformation (CBD) is a powerful and intuitive family of methods that use a coarse mesh, called a cage, to control the deformation of a more detailed inner mesh. CBD relies on cage coordinates to define how points within the cage relate to the cage’s vertices. Various coordinate types, such as mean value coordinates (MVC) [23], [24], harmonic coordinates (HC) [25], [26] and green coordinates (GC) [27], have been developed, all demonstrating strong performance in mesh deformation. Several methods, including DeformingNeRF [2], CageNeRF [3], NeRFShop [4], Li et al. [28], and VolTeMorph [29], have adapted cage-based deformation for radiance fields. These approaches deform sample points along rays during volumetric rendering. However, this strategy is fundamentally incompatible with 3DGS, which uses rasterization, presenting a significant adaptation challenge.

The application of this powerful paradigm to 3DGS, however, remains under-explored. D3GA [30] achieves cage-based deformation on 3DGS human bodies and garments by utilizing tetrahedral mesh cages. Tetrahedral meshes are effective for volumetric deformation, particularly for articulated subjects, but are less common in standard artistic modeling workflows than traditional triangular surface meshes. This leaves an opening for a general-purpose CBD framework that works on arbitrary 3DGS objects using standard, easy-to-edit triangular meshes, and can be extended to variants of 3DGS, which is the focus of this work.

A critical component of a practical CBD workflow is automatic cage construction. For NeRF, methods like NeRFShop [4] and DeformingNeRF [2] employ marching cubes on the opacity field to create a fine mesh, which is then simplified. However, directly interpreting 3DGS as an opacity field is problematic due to noise and artifacts in trained models, particularly floaters. In contrast, our novel cage-generation algorithm adopts a render-reconstruct-simplify approach. By combining T-SDF integration, depth carving, and simplification based on Bounding Proxy [31], our method constructs

high-quality cages not only for standard 3DGS models but also for surface-like variants (e.g., 2DGS [9] or Dai et al. [32]), where marching cubes-based methods would fail, as they cannot be easily converted into opacity fields.

Finally, a key challenge in the mesh simplification process is ensuring the final cage both tightly encloses the object and maintains reasonable deformation coordinates. This requires maintaining non-negative Mean Value Coordinates (MVCs), as negative values can lead to artifacts. Previous methods have struggled to achieve both tightness and non-negative MVC coordinates simultaneously. Approaches like Neural Cage [33] and CageNeRF [3] enforce non-negativity by encouraging simple, convex cages, often at the cost of not being a tight fit that accurately captures the object’s geometry. Conversely, methods that prioritize tightness, such as the original Bounding Proxy technique [31], do not guarantee non-negative MVCs, potentially leading to deformation artifacts. Our approach resolves this dilemma by incorporating the non-negativity of MVCs as a novel two-stage optimization design within the QEM optimization during cage simplification. This ensures the final cage both tightly encloses the object and reduces deformation artifacts caused by negative coordinates.

III. METHOD

A. Preliminaries

3D Gaussian Splatting 3D Gaussian Splatting (3DGS) [1] is a method for representing 3D scenes using a set of 3D Gaussians. Each Gaussian is characterized by its mean $\mu \in \mathbb{R}^3$, covariance $\Sigma \in \mathbb{M}^{3 \times 3}$, opacity $\alpha \in \mathbb{R}$, and color parameters $\mathcal{P} \in \mathbb{R}^k$. The color is view-dependent, modeled using spherical harmonics with k degrees of freedom. The covariance Σ is decomposed as \mathbf{RSSR}^T , where \mathbf{R} is a rotation matrix (encoded as a quaternion) and \mathbf{S} is a scaling matrix (encoded as a scaling vector).

Our approach leverages 3DGS’s key feature: representing scenes as a set of 3D Gaussians, each equivalent to an ellipsoid. This representation forms the foundation of our method.

Cage-based Deformation To deform a fine mesh using a cage, we consider a cage \mathcal{C}_s with vertices $\{\mathbf{v}_j\}$. Points $\mathbf{x} \in \mathbb{R}^3$ inside the cage \mathcal{C}_s can then be represented by cage coordinates $\{\omega_j\}$ (e.g., mean value coordinates [34]). These coordinates define the position of \mathbf{x} relative to the cage vertices. The position of \mathbf{x} is calculated as the weighted sum of cage vertex positions:

$$\mathbf{x} = \sum_j \omega_j \mathbf{v}_j \quad (1)$$

After deforming the cage from \mathcal{C}_s to \mathcal{C}_d with vertices $\{\mathbf{v}'_j\}$, we can compute the new position \mathbf{x}' of \mathbf{x} using the calculated cage coordinates:

$$\mathbf{x}' = \sum_j \omega_j \mathbf{v}'_j \quad (2)$$

The cage defines a continuous field within its boundaries, allowing it to deform the enclosed fine mesh by manipulating

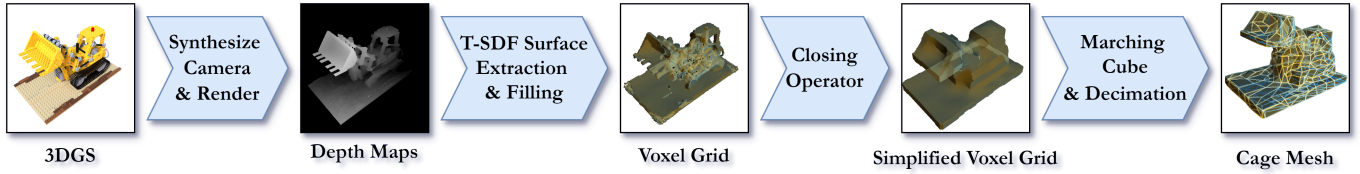


Fig. 1. Overview of our cage-building algorithm. Given an object, our method renders a depth image from it, performs T-SDF integration, surface extraction, and space carving to produce a solid voxel grid. The voxel grid is then simplified using a morphological closing operator, meshed using marching cubes, and decimated to obtain the final cage mesh.

its vertices. This deformation process also works for arbitrary points within the source cage.

Constrained Mesh Decimation Mesh decimation aims to simplify a high-polygon mesh into a low-polygon one. A prominent technique to achieve this is iterative edge collapse, which simplifies the mesh by progressively removing edges from the mesh and merging their two endpoint vertices into a single new vertex. The Quadric Error Metrics (QEM) algorithm [35] is a popular method for guiding this process. It selects which edge to collapse and where to place the new vertex by minimizing a geometric error metric.

However, optimizing for error alone can produce artifacts like self-intersections. To prevent this, QEM can be augmented with constraints. For instance, Bounding Proxy [31] uses a constrained QEM that formulates the placement of the new vertex \mathbf{v} for a collapsed edge (i, j) as a convex quadratic program. The objective is to minimize the sum of squared distances to the planes of incident faces, subject to constraints that prevent local surface inversion:

$$\begin{cases} \min_{\mathbf{v}} E_{ij}(\mathbf{v}) = \mathbf{v}^T \mathbf{Q} \mathbf{v} \\ \text{s.t. } D(\mathbf{v}, f) = \mathbf{n}_f^T (\mathbf{v} - \mathbf{p}_f) \geq 0, \forall f \in \text{EU}(i, j) \end{cases} \quad (3)$$

Here, \mathbf{Q} is the sum of incident plane quadrics, and $\text{EU}(i, j)$ is the set of faces in the 1-ring neighborhood of the edge. The constraint $D(\mathbf{v}, f)$ ensures the new vertex does not move to the wrong side of its neighboring faces, where \mathbf{n}_f is the outward normal of face f .

B. Cage-Building Algorithm

Given a trained 3DGS (or its variants) model \mathcal{S}_s , our automated cage-generation algorithm aims to create a simple cage \mathcal{C}_s that encloses it.

The proposed algorithm revolves around a more robust “render-reconstruct-simplify” pipeline, as illustrated in Figure 1. Most notably, instead of interpreting the 3DGS scene directly as an opacity field, our method extracts the underlying geometry by rendering 2D depth maps from multiple views. Additionally, our final decimation process employs a two-stage process that encourages the non-negativity of internal MVC coordinates, ensuring stable and artifact-free deformations. The full process is broken down into the following steps, following Figure 1:

Synthesize Camera and Render The algorithm begins by synthesizing a set of virtual cameras arranged on a sphere that encompasses the 3DGS scene. Depth maps are then rendered from each of these camera positions. This initial rendering

step serves two crucial purposes. First, it aligns better with the native, rendering-based nature of 3DGS, making the geometry extraction less susceptible to artifacts like low-opacity floaters than methods that treat the scene as an opacity field. Second, by converting the scene into a standardized format of depth maps, the pipeline becomes agnostic to the specifics of the underlying 3DGS representation, ensuring its applicability to both standard 3DGS and its non-volumetric variants.

T-SDF Surface Extraction and Filling The collection of rendered depth maps is then fused into a single, coherent 3D volume using the Truncated Signed Distance Field (T-SDF) integration algorithm from KinectFusion [36]. By integrating depth information from multiple viewpoints, the T-SDF reconstruction naturally averages out inconsistencies and suppresses noise caused by render errors or floaters, resulting in a significantly cleaner surface representation.

To produce a solid volume, this process is followed by a space-carving step. Clean depth maps are re-rendered from the newly created T-SDF volume and used to carve away empty space, filling the interior of the surface. This process yields a high-resolution, solid voxel grid that represents the object’s geometry without the artifacts present in the original 3DGS scene.

Apply Morphological Closing Operator While the reconstructed voxel grid is geometrically accurate, its high level of detail is unsuitable for a deformation cage. To generate a simplified, coarse envelope, a morphological closing operator, as described in Bounding Proxy [31], is applied. This operator is a combination of dilation and erosion, aiming to remove noise and details from the voxel grid, producing a simplified, solid volumetric shape that smoothly envelops the original geometry, making it an ideal precursor to the final cage.

Marching Cube and Decimation In the final stage, we first apply the Marching Cubes algorithm to the simplified voxel grid to extract a dense polygonal mesh. This mesh is then simplified using a constrained edge-collapse algorithm [31] to produce the final low-polygon cage.

Although the standard constrained QEM used in Bounding Proxy [31] produces a cage that tightly encloses the object, many object vertices may still lie outside the cage. This increases the occurrence of negative MVC weights, which in turn causes artifacts in MVC-based deformation. To produce a cage that provides good shape coverage and is optimized for high-quality deformations, our method adopts an alternating two-stage decimation procedure.

Stage 1: Constrained QEM. First, we perform edge collapse using the standard constrained QEM described in the

preliminaries. Each step removes one edge by merging its two endpoints, while strictly enforcing constraints that prevent local surface inversion and ensure the cage remains manifold.

Stage 2: Refinement with MVC Penalty. After a fixed number of collapses, we refine the positions of all cage vertices to reduce negative MVC weights. Specifically, we minimize the following loss:

$$L_{MVC}(V) = \frac{\mu}{|V||S|} \sum_{i \in S} \sum_{\mathbf{v}_j \in V} |\min(\phi_{ji}(V), 0)|^2 + \rho \|V - V_0\|^2$$

where $V = \{\mathbf{v}_j\}$ is the set of all cage vertex positions, $\phi_{ji}(V)$ is the mean value coordinate of a sample point $i \in S$ on the object surface with respect to cage vertex \mathbf{v}_j , and V_0 denotes the positions before refinement. The coefficient $\mu \geq 0$ controls the strength of the MVC penalty, while the regularization term, weighted by $\rho \geq 0$, anchors the vertices near their pre-refinement positions to stabilize the optimization. We minimize L_{MVC} via gradient descent for a fixed number of iterations.

We first run Stage 1 until the vertex count falls below a prescribed threshold, then alternate between the two stages: Stage 1 collapses a number of edges, then Stage 2 optimizes all vertex positions for a number of gradient-descent steps, and so on. This interleaving ensures that the cage remains geometrically valid throughout decimation while progressively improving shape coverage and minimizing negative MVC weights, yielding a cage that is structurally suited for stable, high-quality deformations.

In summary, our method creates high-quality deformation cages through a robust geometry extraction stage followed by a deformation-aware simplification process. By utilizing depth maps and T-SDF integration, we first reconstruct a clean representation of the object, which effectively bypasses 3DGS artifacts like floaters and supports non-volumetric variants of 3DGS. When converting it into a coarse cage, our novel two-stage decimation algorithm optimizes the cage structure. This two-stage process represents a deliberate trade-off: while concave regions necessary for tight enclosure are preserved, the application of the MVC penalty drives negative MVC weights towards zero (see Table VII), resulting in a deformation cage that is not merely a geometric envelope but is structurally optimized for stable, high-fidelity deformation with significantly reduced artifacts.

C. Deformation Algorithm

Our deformation algorithm takes a trained 3DGS scene \mathcal{S}_s along with source and target cages, \mathcal{C}_s and \mathcal{C}_d . These cages define a deformation for part or all of the scene. The objective is to produce the deformed scene \mathcal{S}_d , a 3DGS representation with the specified deformation applied.

The algorithm performs deformation on the 3D Gaussians that constitute the 3DGS scene representation. For each Gaussian s with mean $\mu_s \in \mathbb{R}^3$ and covariance $\Sigma_s \in \mathbb{M}^{3 \times 3}$ (encoded by a rotation matrix \mathbf{R} and scaling matrix \mathbf{S}), our deformation process is applied. This process is illustrated in Figure 2. For a detailed algorithmic description, please refer to the pseudocode provided in Appendix B.

To Ellipsoid To geometrically manipulate the 3D Gaussians, we start by converting them into explicit geometries, namely ellipsoids. The ellipsoids are represented using point clouds so they can be deformed by cage-based deformation.

According to 3DGS [1], a 3D Gaussian with mean μ_s and covariance Σ_s is defined as:

$$G(x) = \exp \left\{ -\frac{1}{2} (\mathbf{x} - \mu_s)^\top \Sigma_s^{-1} (\mathbf{x} - \mu_s) \right\}$$

The ellipsoid is then defined in quadric form as:

$$(\mathbf{x} - \mu_s)^\top \Sigma_s^{-1} (\mathbf{x} - \mu_s) = 1$$

Here, μ_s is the ellipsoid's center \mathbf{c} . The principal axes and their lengths are derived from the eigenvalue decomposition of Σ_s^{-1} , that is:

- The three principal axes (e.g., x, y, z) correspond to the eigenvectors.
- The axis lengths are inversely proportional to the eigenvalues of Σ_s^{-1} .

We define six intersection points of the ellipsoid's principal axes with its surface as:

$$AP = \{\mathbf{c}, \mathbf{x}_1, \mathbf{y}_1, \mathbf{z}_1, \mathbf{x}_2, \mathbf{y}_2, \mathbf{z}_2\} \quad (4)$$

where \mathbf{c} is the center of the ellipsoid, $\mathbf{x}_{1,2}$ denote the two intersection points on the x -axis, with similar definitions for y and z . These points, AP , describe the ellipsoid and form a proxy point cloud for cage-based deformation.

Deform Points with Cages With the proxy point cloud AP_s in place, we apply the desired deformation defined by the source cage \mathcal{C}_s and target cage \mathcal{C}_d to it.

More concretely, we transform AP_s using Mean Value Coordinates (MVC) [23]. First, we convert AP_s from Euclidean coordinates to MVC using \mathcal{C}_s . Then, we convert them back to Euclidean coordinates using \mathcal{C}_d . The resulting deformed axis points are denoted as AP_d .

Transform Gaussians With the original and deformed points, we then estimate the underlying transformation and apply it to the original 3D Gaussians s .

Given the original and deformed points AP_s and AP_d , we compute the transformation $\mathbf{T} \in \mathbb{R}^{3 \times 3}$ by minimizing:

$$\min_{\mathbf{T}} \|\mathbf{D}_d - \mathbf{T}\mathbf{D}_s\|^2 \quad (5)$$

where $\mathbf{D} = [\mathbf{x}_1 - \mathbf{c}, \mathbf{y}_1 - \mathbf{c}, \mathbf{z}_1 - \mathbf{c}, \mathbf{x}_2 - \mathbf{c}, \mathbf{y}_2 - \mathbf{c}, \mathbf{z}_2 - \mathbf{c}] \in \mathbb{R}^{3 \times 6}$ for AP_s or AP_d . The least squares solution for \mathbf{T} is given by:

$$\mathbf{T} = \mathbf{D}_d \mathbf{D}_s^\top (\mathbf{D}_s \mathbf{D}_s^\top)^{-1} \quad (6)$$

The 3D Gaussian s can then be transformed using \mathbf{T} , while using the center of deformed points AP_d as the new mean:

$$\mu_d = \mathbf{c}_d \quad (7)$$

$$\Sigma_d = \mathbf{T} \Sigma_s \mathbf{T}^\top \quad (8)$$

This step is crucial to our algorithm as it updates the Gaussians' covariance, which models their shape and orientation. Not updating covariance would lead to severe artifacts, which we will demonstrate in ablation studies.

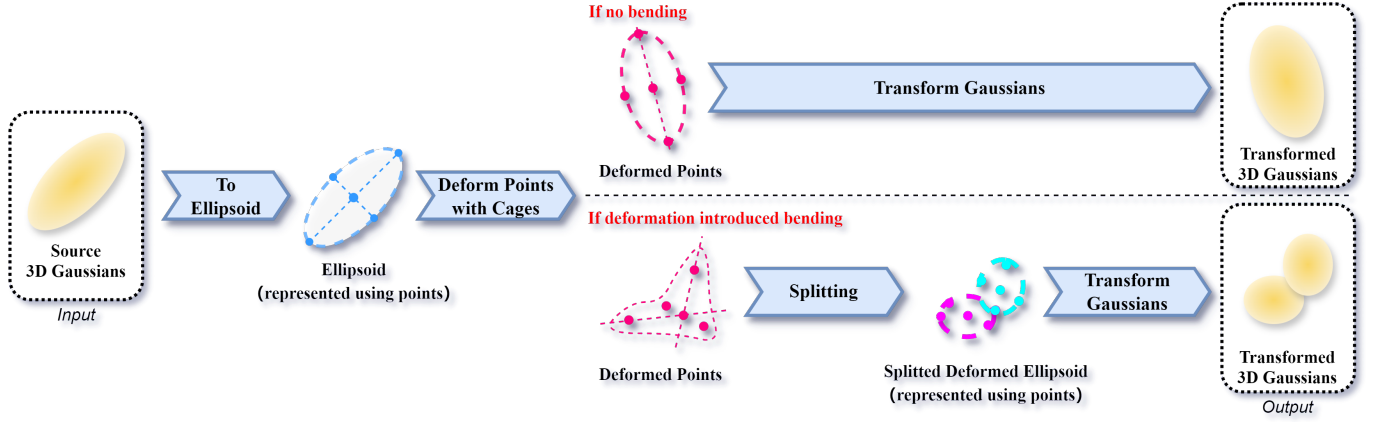


Fig. 2. Overview of our deformation algorithm. The deformation process is shown in 2D for clarity. For deformation, 3DGS Gaussians are converted to ellipsoids represented using points (the proxy point cloud). Proxy points are deformed using cage-based deformation and split if their axes are bent. Finally, deformed points are used to infer transformations for the Gaussians. For more details, please refer to the pseudo-code in Appendix B.

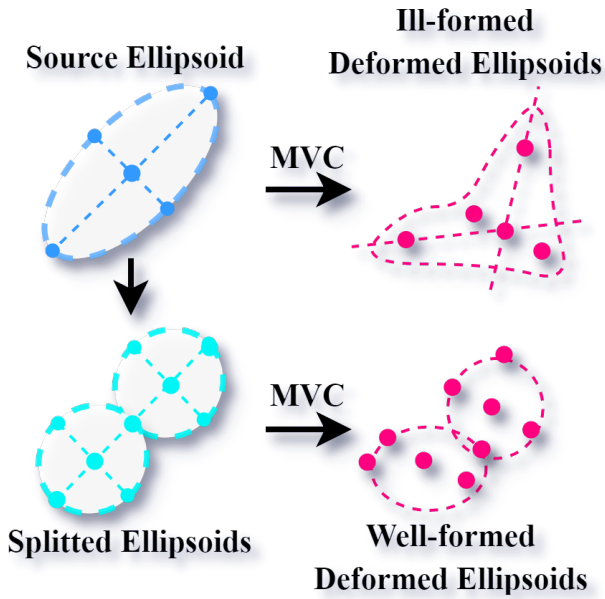


Fig. 3. The splitting process. Our method fixes the ill-formed bent Gaussian by splitting the Gaussian before MVC deformation, leading to well-formed Gaussians and, thus, reasonable transforms.

According to the implementation of 3DGS [1], the transformed mean and covariance can be directly used for rendering. Optionally, to recover the rotation and scaling parameters compatible with 3DGS, we re-fit the potentially sheared covariance matrix Σ_d via eigendecomposition: $\Sigma_d = \mathbf{R}_d \mathbf{\Lambda}_d \mathbf{R}_d^T$. The new rotation is the orthogonal matrix of eigenvectors \mathbf{R}_d , and the new scaling is computed from the square roots of the eigenvalues in $\mathbf{\Lambda}_d$.

Splitting Cage-based deformation enables flexible shape manipulation but introduces non-rigid warping, requiring some Gaussians to bend in deformation. This leads to visual artifacts since a single transformed Gaussian cannot properly represent these bent shapes. To address this, we split the bent Gaussians

into multiple well-formed Gaussians to approximate the bent shapes, as shown in Figure 3.

We start by identifying the bent Gaussians to split. Given a Gaussian with deformed points AP_d , its center is $\mathbf{c}'_d \in AP_d$. The deformed points of an axis (e.g., x-axis) are $\mathbf{x}'_1, \mathbf{x}'_2 \in AP_d$. We calculate the angle between vectors, $\mathbf{x}'_1 - \mathbf{c}'$ and $\mathbf{x}'_2 - \mathbf{c}'$. Splitting is required if the angle falls below a threshold.

The splitting method divides a pre-MVC Gaussian ellipsoid into two smaller ellipsoids along a chosen axis. Each new ellipsoid has its axis lengths reduced by factor k , and both are positioned to meet at the center of the original ellipsoid along the splitting axis. Finally, both resulting ellipsoids undergo MVC transformation again for further processing. The resulting two smaller ellipsoids are more tolerable to deformation along that axis, reducing the risk of bending or distortion when applying MVC directly to elongated shapes. This identification and splitting process is run across all axes to ensure that Gaussians requiring splitting on multiple axes are properly split more than once.

We determine the axis scaling factor k by analyzing the optimal value that can maximally preserve the volume and shape of the original ellipsoid. More concretely, our scaling factor should satisfy two key constraints: Volume Preservation and Shape Preservation.

Volume Preservation Constraint: The volume of the two smaller ellipsoids combined should approximate the volume of the original ellipsoid. This can be expressed as:

$$L_V = \frac{2}{3}\pi abc - 2\frac{2}{3}\pi k_1 k_2 k_3 c \quad (9)$$

where, a, b, c are the semi-principal axes of the original ellipsoid, k_1, k_2, k_3 are the individual scaling factors for the smaller ellipsoids. Simplifying this equation leads to:

$$k_1 k_2 k_3 = \frac{1}{2} \quad (10)$$

Shape Preservation Constraint: Each smaller ellipsoid should maintain a shape similar to the original ellipsoid. This

constraint can be expressed as:

$$\frac{k_1 a}{k_2 b} \approx \frac{a}{b} \quad \frac{k_2 b}{k_3 c} \approx \frac{b}{c} \quad \frac{k_1 a}{k_3 c} \approx \frac{a}{c} \quad (11)$$

From these constraints, the scaling factor for all axes becomes:

$$k_1 = k_2 = k_3 = \sqrt[3]{\frac{1}{2}} \quad (12)$$

Thus, the scaling factor k for splitting should be $\sqrt[3]{\frac{1}{2}}$.

In practice, setting $k = \frac{1}{2}$ produces marginally better results, as demonstrated by our ablation study.

In summary, our algorithm deforms 3DGS by first converting 3D Gaussians into ellipsoids represented using points. Deformations of the points can then be transferred back to the underlying 3D Gaussians. This approach directly operates on the primitives of 3DGS, thus no architectural changes or retraining is needed. Furthermore, with caching, this direct transform approach can work in real-time. Finally, since it operates directly on the primitives, our method can be easily extended to work with 3DGS variants without concerns on training (such as FLoD [37]) and integrate with other 3DGS editing work (such as GaussianEditor [10]) with minimal changes.

Remark We propose a splitting procedure to handle non-rigid warping that bends Gaussians. This approach improves deformation quality for scenes with intricate details, especially when a single Gaussian representing a straight surface must be bent. Unfortunately, the existing methods, such as ARAP-GS [22] and VR-GS [8], ignore this issue.

IV. EXPERIMENTS

A. Implementation Details

For cage building, the voxel grid’s resolution is set to 128. We employ the T-SDF integration and carving procedure from Open3D [38], using their default parameters. For the alternating decimation, in each cycle Stage 1 collapses 10 edges, and Stage 2 runs for 10 gradient-descent steps. We begin alternating at 1000 vertices, sample $|S| = 256$ points on the object surface, and set $\mu = 100$, $\rho = 10^{-4}$. The MVC refinement stage uses the Adam optimizer with a learning rate of 0.005.

For deformation, we set the splitting threshold at 175 degrees and avoid splitting axes with lengths below $1e-2$ for numerical stability. For cages that only encompass part of the scene, we compute the convex hull of the cage and only deform gaussians fall inside it for speed and stability.

We ran all experiments using an NVIDIA A5000 GPU and an AMD Ryzen Threadripper PRO 3975WX processor (32 cores).

B. Cage Building Quality

Our evaluation begins by analyzing the effectiveness and extensibility of our cage-building algorithm.

We evaluate our approach against the traditional marching-cube-then-simplify process used in NeRFShop [4] and DeformingNeRF [2]. However, since the process used by existing

Scene	Method	Mesh Components(↓)	Self-intersections(↓)
Lego	Baseline	2	0
	Ours	1	0
Vase	Baseline	2	3
	Ours	1	0
Vase (2DGS)	Baseline	3	5
	Ours	1	0

TABLE I
QUANTITATIVE COMPARISON ON THE QUALITY OF THE GENERATED CAGE. NOTE THAT THE BASELINE SUFFERS FROM MULTIPLE MESH COMPONENTS (E.G., FLOATERS OR HOLLOW CAVITIES) AND SELF-INTERSECTIONS, WHEREAS OUR METHOD DOES NOT.

methods is designed for radiance fields and cannot directly work with 3DGS, we adapt them based on SuGaR’s [5] approach, as detailed in Appendix C.

We evaluate both methods using the Lego scene from the NeRF Synthetic Dataset [39] and a flower vase extracted from the MipNeRF360 Dataset’s garden scene [40]. To quantitatively evaluate the quality of the generated cages, we also computed their mesh components and the number of intersecting face pairs to detect floaters, hollow cavities, and self-intersections.

In Figure 4, we compare the output of our method with the marching cube-based baseline, showing both the raw voxel grids (before any smoothing operation) and the final meshes. Our method yields cleaner voxel grids for Lego and vase scenes, leading to higher-quality cages. In contrast, the baseline’s voxel grids are significantly noisier and contain more floaters, resulting in self-intersections and floaters in the Lego and vase cages, as shown by Table I.

Our cage-building algorithm also extends naturally to 3DGS variants. When applied to a trained 2DGS representation of the vase (third row of Figure 4 and Table I), the MC baseline method fails to produce a coherent voxel grid before any smoothing, leading to cages with hollow cavities and floaters, while our approach still generates accurate voxel grids and cages suitable for cage-based deformation. This difference occurs because 2DGS uses flat 2D ellipses with minimal volume instead of volumetric 3D ellipsoids. Our method remains effective regardless of this distinction.

C. Deformation Quality

We then compare the deformation quality of our model against existing methods on the NeRF Synthetic Dataset [39]. We start with pre-trained 3D Gaussian Splatting [1] models, apply our cage construction algorithm for cages, manually deform the cages, and run our deformation algorithm. To ensure a direct and fair comparison, we apply the identical deformed cage to all baseline methods. Specifically, for approaches like DeformingNeRF [2], GaMeS [13], SuGaR [5], and Gaussian Frosting [6], we use our cage to deform their underlying mesh or triangle soup, ensuring that any differences in output quality are attributable to the deformation algorithms themselves and not the control input.

Normal Deformations We present our results in Figure 5. In the microphone scene, DeformingNeRF fails to preserve the detailed grid structure of the microphone’s mesh, while Gaussian Frosting produces holes and spiky artifacts on it.

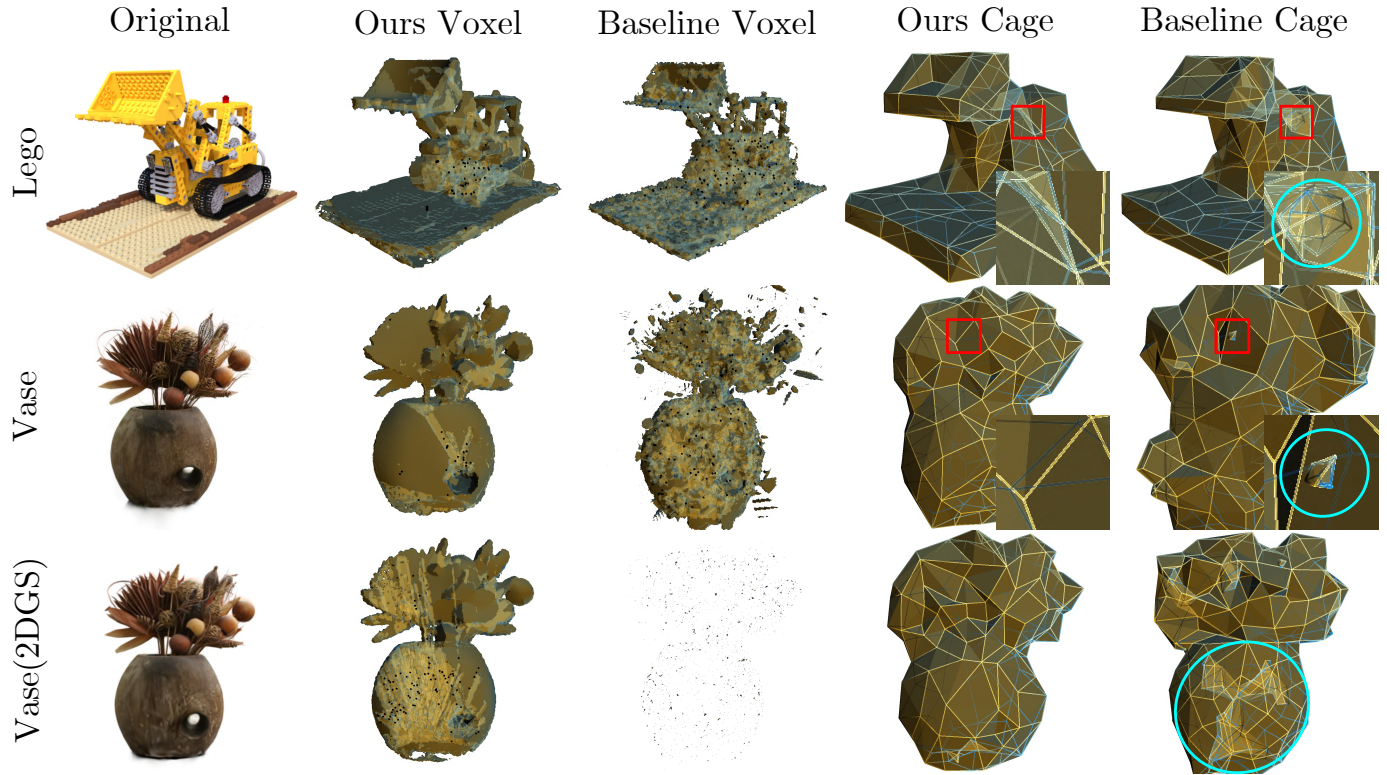


Fig. 4. Comparison of cage building algorithm. We present the raw voxel grids and the produced final cages for comparison. **Red boxes** indicate zoomed areas; **cyan circles** marks defects. Note that our method generates less noisy voxel grids and floater/cavity-free final cages for both 3DGS and 2DGS. The marching cube-based baseline suffers from floaters or hollow cavities, especially on 2DGS, a non-volumetric variant of 3DGS.

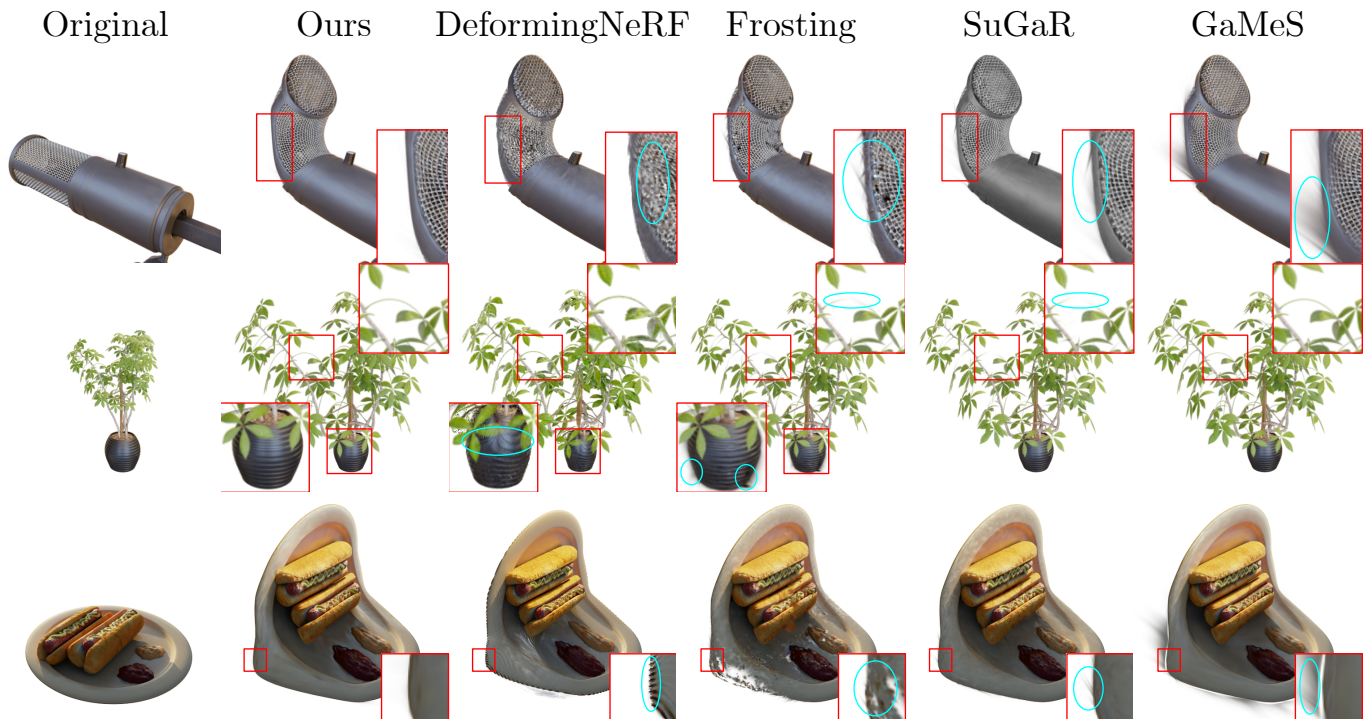


Fig. 5. Comparison of methods on selected objects. All methods shown are driven by the exact same cage for a fair comparison. **Red boxes** indicate zoomed areas; **cyan circles** marks defects. Not having defect marks indicates satisfactory results. Our approach is the only method that performs well across all cases. For more results, refer to our supplementary video.

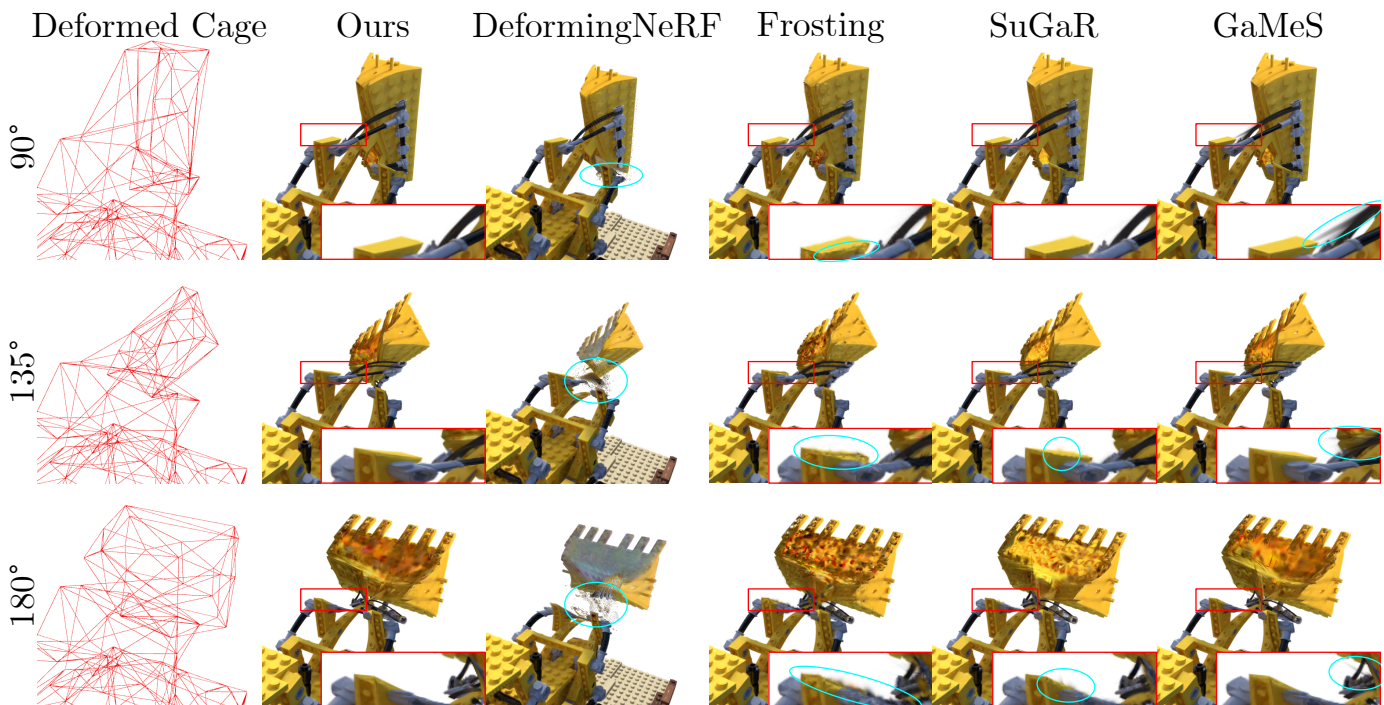


Fig. 6. Comparison of methods from normal to extreme deformations. All methods shown are driven by the exact same cage (shown on left) for a fair comparison. **Red boxes** indicate zoomed areas; **cyan circles** marks defects. Not having defect marks indicates satisfactory results. Note that our method remains robust as deformation intensifies, while other methods develop artifacts. Even under the 180-degree extreme scenario, our method still produces reasonable results.

GaMeS and SuGaR also show spiky artifacts. For the ficus scene, DeformingNeRF and Gaussian Frosting create artifacts on the unedited flower pot. In the expanded upper part, Gaussian Frosting and SuGaR struggle with details, breaking connections between branches. In the hotdog scene, there are wrinkles on the plate with DeformingNeRF, severe tearing with Gaussian Frosting, and spiking artifacts with SuGaR and GaMeS due to the lack of a splitting process. Across all scenes, our approach is the only method that consistently produces the smoothest and most plausible results.

Extreme Deformations We further test how well these methods handle challenging deformations. In Figure 6, we rotate the bulldozer’s head in the Lego scene from the NeRF Synthetic Dataset by 90, 135, and 180 degrees. As can be seen, DeformingNeRF fails to handle twisting. Gaussian Frosting produces blurry artifacts across all angles. GaMeS shows spiky artifacts. While SuGaR works reasonably well at 90 degrees, it creates artifacts at larger angles. In contrast, our method remains stable and generates reasonable results even under extreme rotations.

D. Other Control Methods

Our method controls 3DGS deformation using triangular mesh cages, unlike VR-GS [8], which uses dense tetrahedral grids, and SC-GS [7], which uses sparse control points. Compared to these approaches, our triangular mesh approach is simpler and easier to integrate with existing 3D software or animation pipeline, and it achieves comparable, if not superior, quality in deformation.

Since VR-GS and SC-GS are designed for different scenarios, we adapt them for deformation by directly manipulating their control structure. We evaluate these approaches against our method using three scenes from the D-NeRF [41] dataset: mutant, jumpingjacks, and lego. For fair comparison, we train SC-GS on these dynamic scenes and extract models at a single time frame for testing VR-GS and our method. This is because SC-GS requires training on video data, while other methods do not. We then analyze each method’s control structure for controlling deformation and deformation results.

As shown in Figure 7, our method controls deformation using standard triangle meshes, while VR-GS uses dense tetrahedral grids and SC-GS uses sparse control points. Our approach offers a simpler structure than VR-GS’s simulation-focused design, and integrates better with existing animation software than both SC-GS and VR-GS. In terms of result quality, our method demonstrates clear advantages over SC-GS, producing smoother shoulder deformations in the jumpingjacks and mutant scenes, while avoiding the disconnection artifacts present in the lego scene. When compared to VR-GS, our method produces smoother details on the lego and jumpingjacks scenes, and the results are comparable in the mutant scene. This shows that our method is more effective in handling complex deformations while using a simpler and easier-to-integrate control method.

E. Training & Deformation Speed

We then benchmark the training and deformation times across all methods. We test on two sets of scenes/cages:

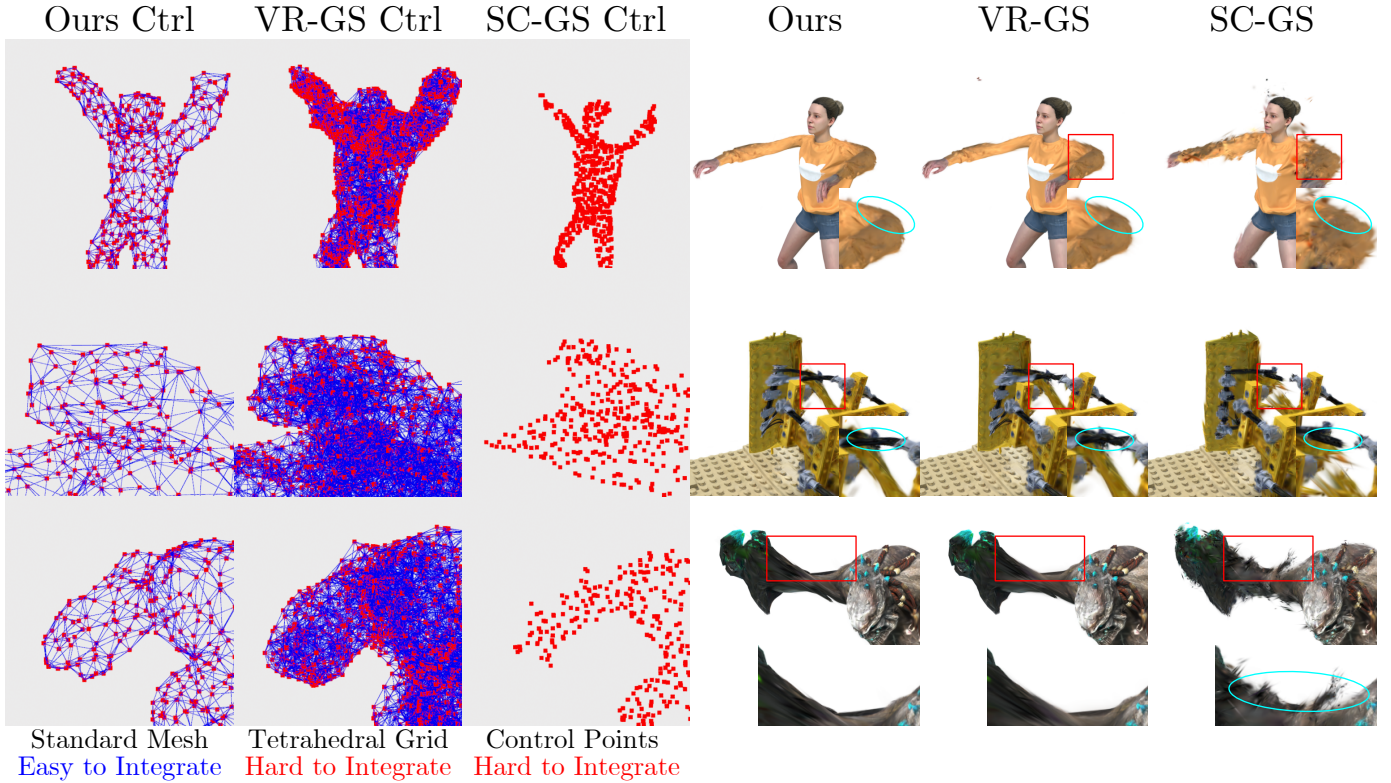


Fig. 7. Comparison with methods that can be adapted for deformation. We show the method’s control structures and their output. **Cyan circles** mark defects, not having cyan circles indicates satisfactory results. Our method uses triangle mesh cages for control, which is simpler and easier to integrate with existing animation software such as Blender, unlike VR-GS or SC-GS. Our method also produces superior results. While VR-GS performed adequately on the mutant scene, it struggled with the lego scene’s detailed deformations.

NeRF Scenes/Cages We select scenes from the NeRF Synthetic Dataset [39] and use the cages from our cage construction algorithm for deformation. The selected scenes are: lego, chair, ficus, and hotdog.

DeformingNeRF Scenes/Cages We also test on the scenes selected by DeformingNeRF [2], using DeformingNeRF’s cages as well. More concretely, DeformingNeRF [2] selected two scenes from the NeRF Synthetic Dataset [39](chair and lego) and two from the NSVF [42] Synthetic Dataset (robot and toad).

We chose the ficus and hotdog scene over the robot and toad scene to test the method’s performance when scaling objects with significant details or splitting Gaussians representing flat surfaces. In terms of cages, our cages are automatically generated and undergo extensive deformations, while DeformingNeRF’s cages are manually created and have milder deformations.

Training Speed Table II presents the average training time for all methods, while the detailed per-scene results are shown in Table III. Training times of vanilla 3DGS are provided for reference. With a pre-trained vanilla 3DGS model or its variants, our method can directly deform it without retraining or conversion, hence no training would be needed. In contrast, other approaches require retraining or conversion because they altered the architecture of 3DGS for editability.

Deformation Speed Table IV presents the average deformation time for all methods, and the detailed per-scene

Method	NeRF [39] Scenes training (sec.↓)	DeformingNeRF [2] Scenes training (sec.↓)
DeformingNeRF [2]	491.33	479.18
SuGaR [5]	3234.30	3166.88
GaMeS [13]	432.02	469.65
Frosting [6]	2649.78	2673.52
Vanilla 3DGS [1]	460.19	462.38
Ours	N/A	N/A

TABLE II
BENCHMARK RESULTS COMPARING TRAINING TIMES. **RED** INDICATES BEST VALUES, **BLUE** MARKS SECOND-BEST. NOTE THAT WITH A PRE-TRAINED VANILLA 3DGS MODEL, OUR METHOD CAN DIRECTLY DEFORM IT WITHOUT RETRAINING OR CONVERSION.

results are provided in Table V and Table VI. As Table IV presents, our method, on average, achieves 60FPS on the simpler DeformingNeRF cages and our more challenging cages, hence real-time deformation. In terms of once-per-scene preprocessing, our approach is slower compared to mesh-based 3DGS methods like SuGaR [5], GaMeS [13], and Gaussian Frosting [6]. This is because our method needs to process more points for deformation and splitting, while mesh-based methods can simply deform the underlying mesh or triangle soup. However, our method is the fastest in rendering, as we use the unmodified vanilla 3DGS rendering process.

This can also be seen from the per-scene results shown in Table V and Table VI. As can be seen, our method is consistently the second-fastest in deformation and fastest in

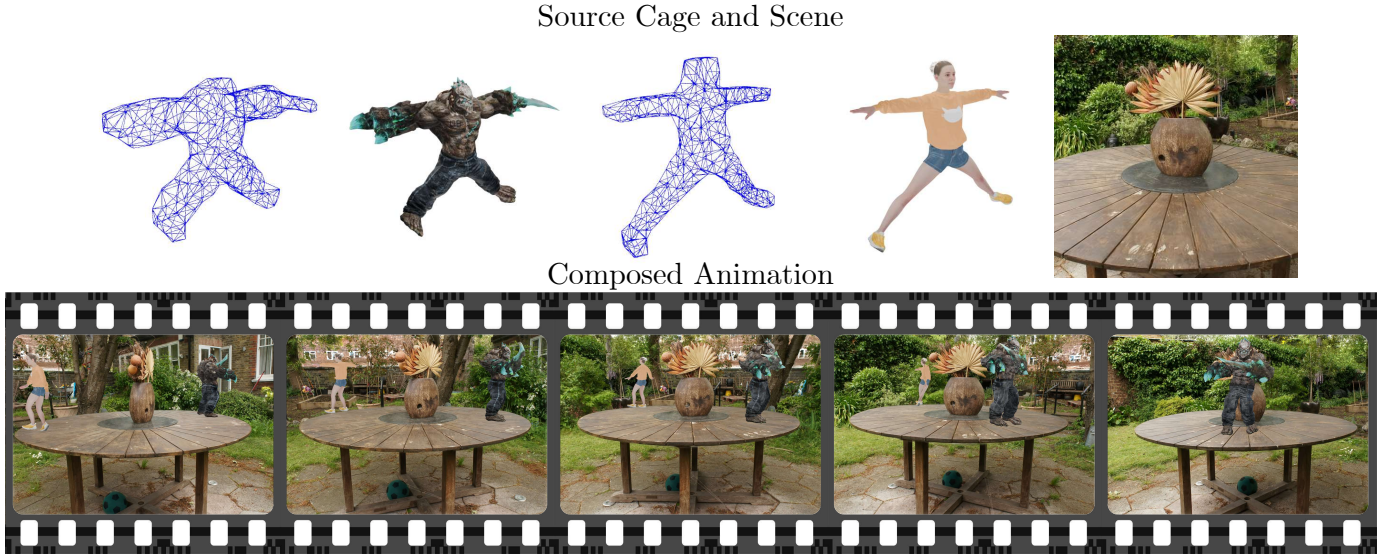


Fig. 8. Animating and composing scenes using our method. Note that our cage-based deformation algorithm can be used for animation and composition by animating the cages and placing the cages into another scene. Please refer to our supplementary video for more details.

Scene	NeRF [39] Scenes		DeformingNeRF [2] Scenes		
	Method	training (sec.↓)	Scene	Method	training (sec.↓)
chair	DeformingNeRF [2]	451.15	chair	DeformingNeRF [2]	451.15
	SuGaR [5]	3154.14		SuGaR [5]	3154.14
	GaMeS [13]	427.86		GaMeS [13]	427.86
	Frosting [6]	2651.98		Frosting [6]	2651.98
	Vanilla 3DGS [1]	417.16		Vanilla 3DGS [1]	417.16
Ours	N/A	Ours	N/A		
lego	DeformingNeRF [2]	515.22	lego	DeformingNeRF [2]	515.22
	SuGaR [5]	3264.54		SuGaR [5]	3264.54
	GaMeS [13]	462.17		GaMeS [13]	462.17
	Frosting [6]	2726.72		Frosting [6]	2726.72
	Vanilla 3DGS [1]	457.76		Vanilla 3DGS [1]	457.76
Ours	N/A	Ours	N/A		
hotdog	DeformingNeRF [2]	621.38	robot	DeformingNeRF [2]	439.98
	SuGaR [5]	3475.71		SuGaR [5]	3060.61
	GaMeS [13]	401.03		GaMeS [13]	407.78
	Frosting [6]	2606.45		Frosting [6]	2603.48
	Vanilla 3DGS [1]	535.54		Vanilla 3DGS [1]	400.25
Ours	N/A	Ours	N/A		
figus	DeformingNeRF [2]	377.58	toad	DeformingNeRF [2]	510.36
	SuGaR [5]	3042.79		SuGaR [5]	3188.21
	GaMeS [13]	437.01		GaMeS [13]	580.80
	Frosting [6]	2613.99		Frosting [6]	2711.88
	Vanilla 3DGS [1]	430.30		Vanilla 3DGS [1]	574.36
Ours	N/A	Ours	N/A		

TABLE III

DETAILED PER-SCENE TRAINING TIME OF TABLE II. **RED** INDICATES BEST VALUES, **BLUE** MARKS SECOND-BEST. TRAINING TIMES OF VANILLA 3DGS ARE ALSO PROVIDED FOR REFERENCE.

rendering.

V. APPLICATIONS

Our method can be extended to work with other methods or 3DGS variants, making it valuable for many application scenarios.

A. Composition and Animation

Since our deformation algorithm enables cage-based deformation on 3DGS, we can animate and compose different



Fig. 9. Integrating our method with GaussianEditor [10]. Note that our approach successfully stretched the nose of the face edited by GaussianEditor. Please refer to our demo video for more results.

3DGS models by animating their cages and placing the cages into another scene. This animation and composition can be done using existing mesh editing and animation software, such as Blender [43].

Figure 8 shows animated 3DGS versions of Mixamo [44] characters Sophie and Mutant, composited into the MipNeRF360 [40] garden scene. This is done by animating the cage of the characters using the automatic rigging and animation functionality of Mixamo, placing the cages within the garden scene, and applying our algorithm. As can be seen, our model achieves high-quality animation and composition of 3DGS models, which is useful for animators and artists intending to work with 3DGS.

B. Combining with Editing Methods

Our deformation algorithm achieves low-level, direct shape editing and can be integrated with other work on editing or editable 3DGS to complement their editing abilities.

We start by integrating with GaussianEditor [10], a text-prompt-based 3DGS editing method. As shown in Figure 9, in the face scene from the Instruct-NeRF2NeRF Dataset [46], we use GaussianEditor to select the scene’s facial region and edit

Method	NeRF [39] Scenes (Automatic Cages)			DeformingNeRF [2] Scenes (Manual Cages)		
	preprocess (ms↓)	deform (ms↓/FPS↑)	render (ms↓/FPS↑)	preprocess (ms↓)	deform (ms↓/FPS↑)	render (ms↓/FPS↑)
DeformingNeRF* [2]	3530.10	3933.93 / 0.25FPS	4970.13 / 0.20FPS	2642.48	2420.32 / 0.41FPS	3441.58 / 0.29FPS
SuGaR [5]	1197.03	1483.37 / 0.67FPS	17.26 / 57.94FPS	746.24	1474.52 / 0.68FPS	16.78 / 59.59FPS
GaMeS [13]	1517.75	8.34 / 119.90FPS	5.87 / 170.36FPS	1183.13	6.56 / 152.44FPS	6.24 / 160.26FPS
Frosting [6]	1163.60	125.83 / 7.95FPS	17.27 / 57.90FPS	715.36	122.48 / 8.16FPS	16.62 / 60.17FPS
Ours	3565.11	16.42 / 60.90FPS	4.12 / 242.72FPS	2744.05	12.33 / 81.10FPS	3.97 / 251.89FPS

* DeformingNeRF performs deformation during rendering. Note that the render time involves the deformation time.

TABLE IV

BENCHMARK RESULTS COMPARING DEFORMATION TIMES. **RED** INDICATES BEST VALUES, **BLUE** MARKS SECOND-BEST. DEFORMATION TIMES INCLUDE ONCE-PER-SCENE PREPROCESSING AND ACTUAL DEFORMATION. THE TIME TO RENDER THE DEFORMED REPRESENTATION IS ALSO PRESENTED HERE. OUR METHOD ACHIEVES REAL-TIME PERFORMANCE (~ 60 FPS) FOR BOTH CAGE TYPES AND IS THE FASTEST IN RENDERING.

Scene	Method	NeRF [39] Scenes		
		preprocess (ms↓)	deform (ms↓/FPS↑)	render (ms↓/FPS↑)
chair	DeformingNeRF [2]	3361.99	2882.62 / 0.35FPS	3908.45 / 0.26FPS
	SuGaR [5]	1055.86	1528.79 / 0.65FPS	17.04 / 58.69FPS
	GaMeS [13]	1355.31	7.25 / 137.93FPS	5.42 / 184.50FPS
	Frosting [6]	1023.09	125.62 / 7.96FPS	16.36 / 61.12FPS
	Ours	3124.88	13.89 / 71.99FPS	3.43 / 291.55FPS
lego	DeformingNeRF [2]	4075.17	4633.86 / 0.22FPS	5668.55 / 0.18FPS
	SuGaR [5]	1510.96	1511.84 / 0.66FPS	16.91 / 59.14FPS
	GaMeS [13]	2188.09	12.09 / 82.71FPS	6.43 / 155.52FPS
	Frosting [6]	1495.65	133.17 / 7.51FPS	16.10 / 62.11FPS
	Ours	5148.76	23.71 / 42.18FPS	4.12 / 242.72FPS
hotdog	DeformingNeRF [2]	3500.92	3989.47 / 0.25FPS	5028.07 / 0.20FPS
	SuGaR [5]	1234.86	1477.10 / 0.68FPS	17.46 / 57.27FPS
	GaMeS [13]	803.10	4.43 / 225.73FPS	4.30 / 232.56FPS
	Frosting [6]	1156.40	122.38 / 8.17FPS	17.01 / 58.79FPS
	Ours	1918.72	9.53 / 104.93FPS	3.25 / 307.69FPS
ficus	DeformingNeRF [2]	3182.31	4229.77 / 0.24FPS	5275.44 / 0.19FPS
	SuGaR [5]	986.44	1415.76 / 0.71FPS	17.61 / 56.79FPS
	GaMeS [13]	1724.51	9.59 / 104.28FPS	7.34 / 136.24FPS
	Frosting [6]	979.27	122.16 / 8.19FPS	19.61 / 50.99FPS
	Ours	4068.09	18.53 / 53.97FPS	5.68 / 176.06FPS

TABLE V

DETAILED PER-SCENE DEFORMATION TIME OF TABLE IV ON NeRF SCENES. NOTE THAT OUR METHOD IS CONSISTENTLY THE SECOND-FASTEST IN DEFORMATION AND FASTEST IN RENDERING.

Scene	Method	DeformingNeRF [2] Scenes		
		preprocess (ms↓)	deform (ms↓/FPS↑)	render (ms↓/FPS↑)
chair	DeformingNeRF [2]	3422.40	2910.13 / 0.34FPS	3936.68 / 0.25FPS
	SuGaR [5]	1061.50	1486.58 / 0.67FPS	16.91 / 59.14FPS
	GaMeS [13]	1353.42	7.49 / 133.51FPS	5.43 / 184.16FPS
	Frosting [6]	1018.24	126.04 / 7.93FPS	16.98 / 58.89FPS
	Ours	3139.67	14.31 / 69.88FPS	3.71 / 269.54FPS
lego	DeformingNeRF [2]	2084.41	1197.90 / 0.83FPS	2219.81 / 0.45FPS
	SuGaR [5]	441.99	1532.87 / 0.65FPS	16.83 / 59.42FPS
	GaMeS [13]	593.17	3.36 / 297.62FPS	6.17 / 162.07FPS
	Frosting [6]	424.85	125.03 / 8.00FPS	15.84 / 63.13FPS
	Ours	1378.66	6.18 / 161.81FPS	3.94 / 253.81FPS
robot	DeformingNeRF [2]	1895.59	1441.66 / 0.69FPS	2440.74 / 0.41FPS
	SuGaR [5]	517.81	1421.84 / 0.70FPS	16.43 / 60.86FPS
	GaMeS [13]	706.71	3.86 / 259.07FPS	5.68 / 176.06FPS
	Frosting [6]	510.92	117.94 / 8.48FPS	17.00 / 58.82FPS
	Ours	1664.74	7.19 / 139.08FPS	3.94 / 253.81FPS
toad	DeformingNeRF [2]	3167.54	4131.60 / 0.24FPS	5169.09 / 0.19FPS
	SuGaR [5]	963.66	1456.78 / 0.69FPS	16.95 / 59.00FPS
	GaMeS [13]	2079.23	11.51 / 86.88FPS	7.69 / 130.04FPS
	Frosting [6]	907.43	120.92 / 8.27FPS	16.67 / 59.99FPS
	Ours	4793.12	21.63 / 46.23FPS	4.29 / 233.10FPS

TABLE VI

DETAILED PER-SCENE DEFORMATION TIME OF TABLE IV ON DeformingNeRF SCENES. NOTE THAT OUR METHOD IS CONSISTENTLY THE SECOND-FASTEST IN DEFORMATION AND FASTEST IN RENDERING.

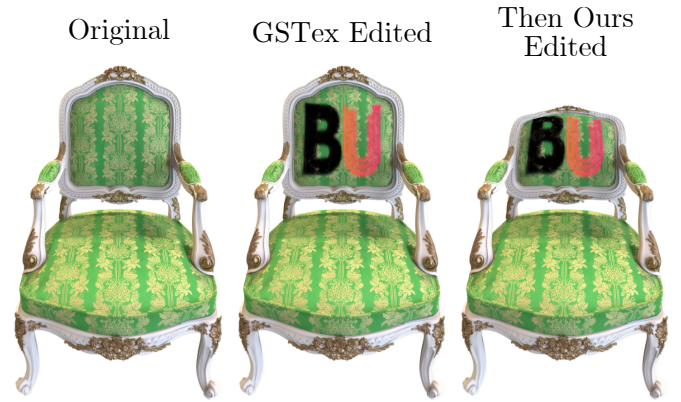


Fig. 10. Integrating our method with GSTex [45]. Note that GSTex enables texture editing, and our method enables shape editing combined to provide comprehensive editing capabilities.

its appearance with the text prompt "turn him into a clown". We then apply our deformation algorithm to stretch the nose of the edited face. As can be seen, our method successfully stretched the nose of the edited face.

Our method can also be integrated with GSTex [45], a 2DGS variant that allows texture editing. We tested this integration using a chair from the NeRF Synthetic Dataset [39]. As shown in Figure 10, we modified the chair's texture using GSTex, then applied our deformation algorithm to deform the result. This combination demonstrates how our approach enhances the overall editing capabilities.

C. Extending to Variants of 3DGS

Our method can also be extended to work with other 3DGS variants.

We start by integrating with 2DGS [9], a method that uses flattened 2D Gaussian disks instead of 3D Gaussian balls for improved geometry, depth rendering, and normal reconstruction. We demonstrate this by deforming a 2DGS capturing the lego scene from the NeRF Synthetic Dataset [39]. Results are shown in Figure 11; note the depth and normal render of the deformed model is high-quality as well.

We also integrate with FLoD [37], a method that adds Level-of-Details(LoD) to 3DGS. We test this by editing a FLoD capturing the garden scene from the MipNeRF360 dataset [40]. As shown by Figure 12, our deformation works well across

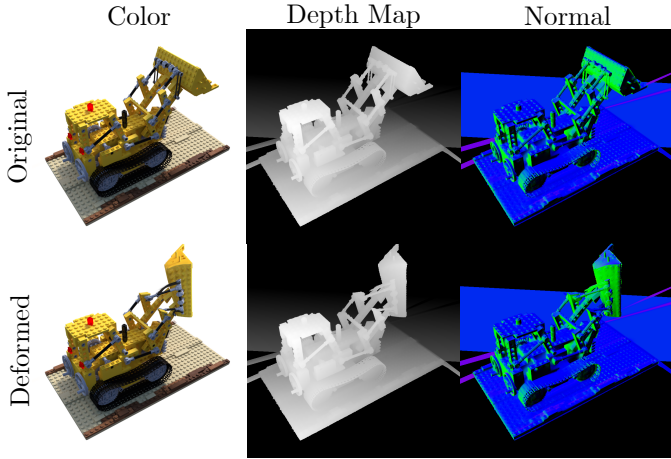


Fig. 11. Integrating our method with 2DGS [9]. Note that the deformation performed by our method is not only high-quality in RGB rendering but also in depth and normal map as well.

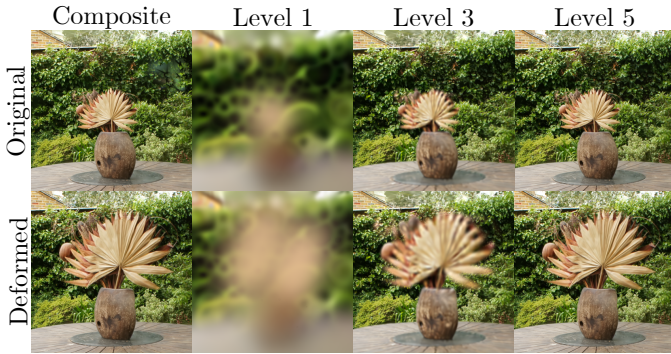


Fig. 12. Integrating our method with FLoD [37]. Our method enlarged the flowers, which is correctly applied to all Level-of-Details(LoD) levels and the final composite result.

all LoDs, demonstrating our method’s adaptability to these variants of 3DGS. This adaptability could allow our method to utilize the available information across different models for different applications.

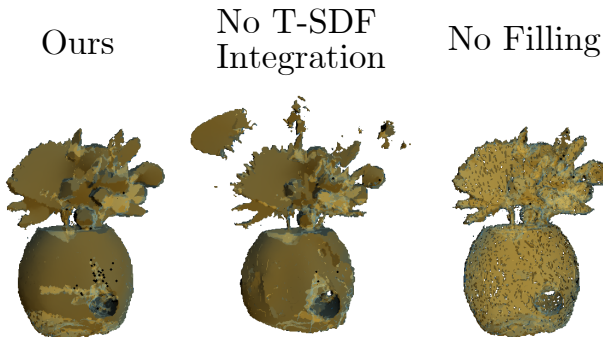


Fig. 13. Ablation study results for cage building algorithm. The voxel grids before closing simplification are shown. Note that not performing T-SDF integration would lead to artifacts on top of the vase, while not performing filling leads to porous and hollow shapes. Both would harm cage quality.

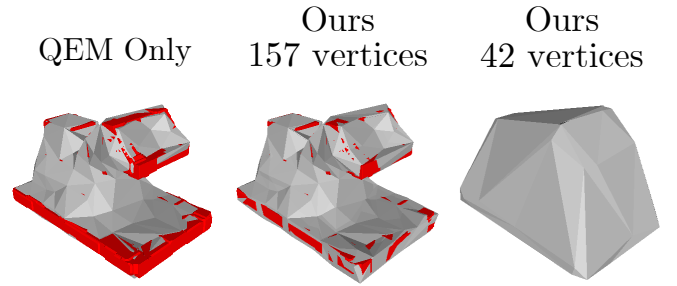


Fig. 14. Ablation study results for our two-stage mesh simplification process. Note how our two-stage method produces a cage (gray) that better envelops the object geometry (red) compared to the standard QEM approach. Further simplification perfects the enclosure but at the cost of detail, demonstrating a trade-off.

	Neg. MVC Entry	Smallest Neg. MVC Value
Improvement	8.7%	99.67%

TABLE VII

MVC STATISTICS CHANGE FOR THE LEGO SCENE. OUR TWO-STAGE METHOD MAKES 8.7% OF MVC ENTRIES NON-NEGATIVE AND REDUCES THE MAGNITUDE OF THE SMALLEST NEGATIVE WEIGHT BY 99.67%.

D. Ablation Studies

Cage-building Algorithm We start our ablation study by analyzing the key steps in our cage-building algorithm, using the vase segmented from the garden scene in the MipNeRF360 [40] dataset. We compare our voxel grid extraction method against two baselines: a naive approach that skips T-SDF integration and directly constructs voxel grids by performing depth carving using 3DGS-rendered depth maps (“No T-SDF Integration”), and a version of our algorithm that skips depth carving and use the surface points extracted from T-SDF integration instead (“No Filling”).

Figure 13 shows the produced voxel grids prior to applying the closing operator. Using depth maps directly for voxel carving without T-SDF integration creates noise on the vase’s top. Without filling, the resulting voxel grid is porous and hollow, which closing operations cannot effectively remedy. Both approaches produce suboptimal results and could harm the cage’s quality.

Two-Stage Mesh Simplification We now validate our two-stage mesh simplification design, which is critical for generating high-quality cages. We compare our full algorithm against a baseline using only the first stage, namely Constrained QEM, on the lego scene from the NeRF Synthetic dataset [39].

We further quantify this comparison on the lego scene. Using the centers of Gaussian ellipsoids as geometry proxies, we evaluate their MVC weights on the cage before and after decimation. We then measure the percentage point drop in negative MVC entries and the relative change in the smallest negative (worst-case) MVC weight, reporting both in Table VII.

The results are visualized in Figure 14. As can be seen, the baseline QEM Only approach produces a cage that encloses the object too tightly, leaving significant portions of the dense mesh (shown in red) outside the cage. In contrast, our two-stage method (middle, 157 vertices) generates a cage that provides superior coverage. The inclusion of the MVC penalty term encourages the cage to properly envelop the object,

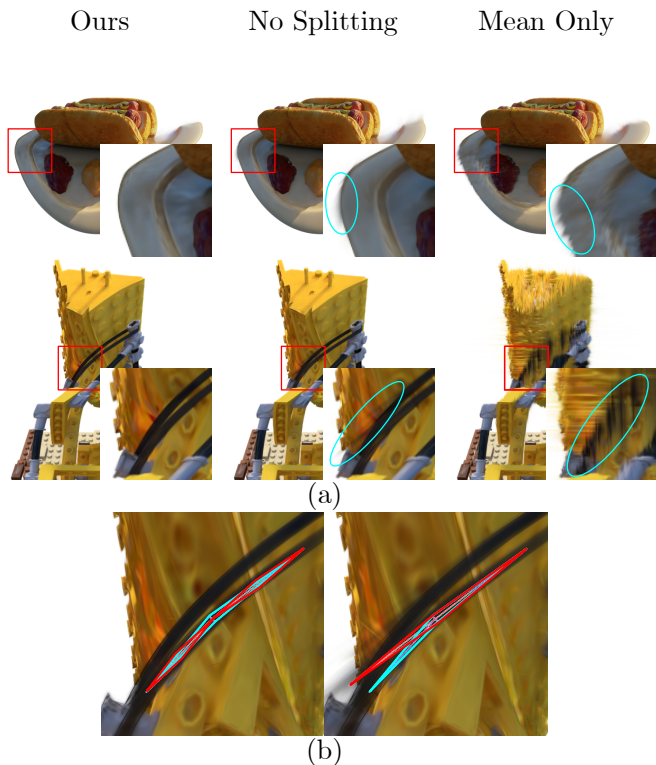


Fig. 15. (a) Ablation study results for deformation algorithm. Note the sharp spikes caused by disabling splitting in the highlighted area. The naive mean-only variant produces significant artifacts as well. (b) The zoomed-in view of (a) highlights the deformed Gaussian (cyan diamond) and the actual Gaussians (red diamond), including both splitting and non-splitting Gaussians). Note that when the deformed Gaussian is approximated using only one Gaussian, rather than splitting, noticeable spiking artifacts appear.

while also decreasing the occurrence of negative MVC weights for interior points. This confirms that our two-stage design effectively produces cages that are structurally optimized for high-quality, stable deformations. While further simplifying the cage (right) eliminates unenclosed regions entirely and further reduces negative MVC weights, it causes a significant loss of detail, highlighting a trade-off between cage quality for deformation and its fidelity to the object’s original shape.

The effectiveness of our design can also be seen from Table VII. Our method makes 8.7% of MVC entries non-negative and reduces the magnitude of the smallest negative MVC value by 99.67%. Showcasing its ability to significantly drive negative MVC weights toward zero.

Deformation Algorithm To evaluate our design choices in the deformation algorithm, we compare our algorithm with two simpler variants: one without splitting and another that directly applies cage-based deformation to the position(mean) of the Gaussians. We test these algorithms on the lego and hotdog scenes from the NeRF Synthetic Dataset [39]. Figure 15 shows the results.

As can be seen, the simpler mean-only variant of our algorithm produced significant striping artifacts. This is prevented in our algorithm by transforming the covariance (thus, rotation and shape) of the Gaussians as well. Furthermore, as shown in the highlighted and zoomed-in regions, using our algorithm without splitting results in spiking artifacts in bent regions.

This occurs because the long belt is represented by a few thin, elongated ellipsoids. When bent, noticeable spiking artifacts appear, as shown in Figure 15 (b). Clearly, bending is challenging to approximate using a single Gaussian ellipsoid instead of splitting. This demonstrates that the splitting procedure is essential for handling large deformations. Simply optimizing the existing Gaussians, as proposed by ARAP-GS [22] or VR-GS [8], is insufficient for accurately modeling bending.

Split Scaling Factor Finally, we analyze our choice of scaling factor k in the splitting step using the Lego scene from the NeRF Synthetic Dataset [39].

As shown in Figure 16, the scaling factor k significantly impacts results. A large k (e.g., 1.0) creates spikes on the bulldozer’s belt due to oversized split Gaussians failing to model the intricate bending deformation. Conversely, a small k (e.g., 0.1) produces fragmented surfaces as the split Gaussians become too small to cover the belt. While our theoretical analysis suggests k should be $\sqrt[3]{\frac{1}{2}}$ (denoted as $\text{cbrt}(0.5)$), empirically we find $\frac{1}{2}$ produces superior results.

VI. CONCLUSION

In this paper, we introduced GSDeformer, a cage-based deformation algorithm for 3D Gaussian Splatting (3DGS). Our approach can directly deform existing trained vanilla 3DGS in real time and can be easily extended to its variants. We adapt cage-based deformation for 3DGS by first building a proxy point cloud from the Gaussians and then transferring the point cloud’s deformation back to 3DGS, splitting the relevant Gaussians to handle bending. This approach requires no additional training data or architectural changes. We also developed an algorithm that automatically constructs cages for 3DGS deformation, featuring a two-stage simplification process that encourages the cage to better envelope the object and be structured for stable high-quality deformations.

Limitations Currently, our algorithm simply copies the spherical harmonics parameters for viewpoint-dependent color, without accounting for the effect of rotation. Additionally, compared to cage-based deformation, point-based deformation is better suited for creating smaller deformations, such as facial expressions. Extending 3DGS to capture and edit facial microexpressions is part of our future work.

REFERENCES

- [1] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, “3d gaussian splatting for real-time radiance field rendering,” *ACM Transactions on Graphics*, vol. 42, no. 4, July 2023. [Online]. Available: <https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/>
- [2] T. Xu and T. Harada, “Deforming radiance fields with cages,” in *ECCV*, 2022.
- [3] Y. Peng, Y. Yan, S. Liu, Y. Cheng, S. Guan, B. Pan, G. Zhai, and X. Yang, “Cagenerf: Cage-based neural radiance fields for generalized 3d deformation and animation,” in *Thirty-Sixth Conference on Neural Information Processing Systems*, 2022.
- [4] C. Jambon, B. Kerbl, G. Kopanas, S. Diolatzis, T. Leimkühler, and G. Drettakis, “Nerfshop: Interactive editing of neural radiance fields,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 6, no. 1, May 2023. [Online]. Available: <https://repo-sam.inria.fr/fungraph/nerfshop/>
- [5] A. Guédon and V. Lepetit, “Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering,” *arXiv preprint arXiv:2311.12775*, 2023.

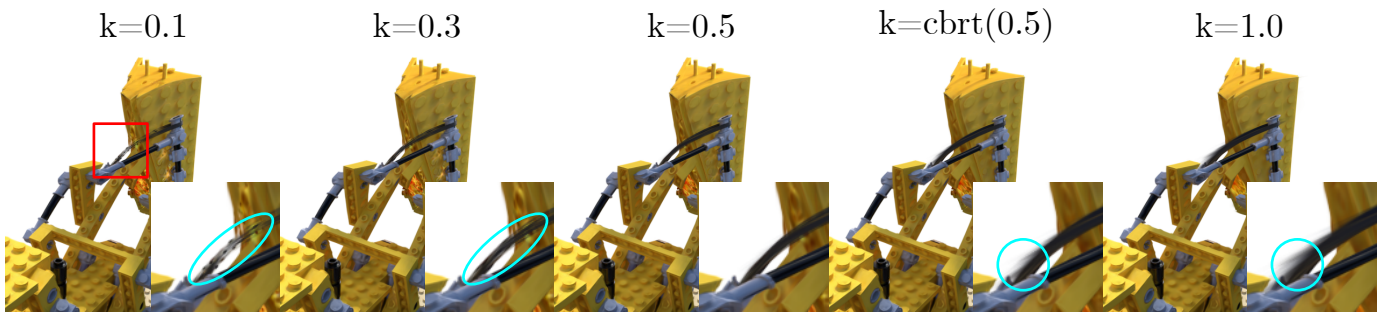


Fig. 16. Ablation study results for the split factor k . Note that setting k too low creates fragmented surfaces, while setting it too high results in sharp spikes. It is shown that 0.5 achieves a better balance than our theoretically derived value.

- [6] —, “Gaussian frosting: Editable complex radiance fields with real-time rendering,” *arXiv preprint arXiv:2403.14554*, 2024.
- [7] Y.-H. Huang, Y.-T. Sun, Z. Yang, X. Lyu, Y.-P. Cao, and X. Qi, “Sc-gs: Sparse-controlled gaussian splatting for editable dynamic scenes,” *arXiv preprint arXiv:2312.14937*, 2023.
- [8] Y. Jiang, C. Yu, T. Xie, X. Li, Y. Feng, H. Wang, M. Li, H. Lau, F. Gao, Y. Yang, and C. Jiang, “Vr-gs: A physical dynamics-aware interactive gaussian splatting system in virtual reality,” *arXiv preprint arXiv:2401.16663*, 2024.
- [9] B. Huang, Z. Yu, A. Chen, A. Geiger, and S. Gao, “2d gaussian splatting for geometrically accurate radiance fields,” in *SIGGRAPH 2024 Conference Papers*. Association for Computing Machinery, 2024.
- [10] Y. Chen, Z. Chen, C. Zhang, F. Wang, X. Yang, Y. Wang, Z. Cai, L. Yang, H. Liu, and G. Lin, “Gaussianeditor: Swift and controllable 3d editing with gaussian splatting,” *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21 476–21 485, 2023.
- [11] Y. Wang, X. Yi, Z. Wu, N. Zhao, L. Chen, and H. Zhang, “View-consistent 3d editing with gaussian splatting,” *ArXiv*, vol. abs/2403.11868, 2024.
- [12] J. Wu, J. Bian, X. Li, G. Wang, I. D. Reid, P. Torr, and V. A. Prisacariu, “Gausctrl: Multi-view consistent text-driven 3d gaussian splatting editing,” *ArXiv*, vol. abs/2403.08733, 2024.
- [13] J. Waczyńska, P. Borycki, S. Tadeja, J. Tabor, and P. Spurek, “Games: Mesh-based adapting and modification of gaussian splatting,” 2024.
- [14] L. Gao, J. Yang, B.-T. Zhang, J. Sun, Y.-J. Yuan, H. Fu, and Y.-K. Lai, “Mesh-based gaussian splatting for real-time large-scale deformation,” *ArXiv*, vol. abs/2402.04796, 2024.
- [15] X. Gao, X. Li, Y. Zhuang, Q. Zhang, W. Hu, C. Zhang, Y. Yao, Y. Shan, and L. Quan, “Mani-gs: Gaussian splatting manipulation with triangular mesh,” *arXiv preprint arXiv:2405.17811*, 2024.
- [16] P. Wang, L. Liu, Y. Liu, C. Theobalt, T. Komura, and W. Wang, “Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction,” *NeurIPS*, 2021.
- [17] T. Xie, Z. Zong, Y. Qiu, X. Li, Y. Feng, Y. Yang, and C. Jiang, “Physgaussian: Physics-integrated 3d gaussians for generative dynamics,” *arXiv preprint arXiv:2311.12198*, 2023.
- [18] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle, “The material point method for simulating continuum materials,” in *ACM SIGGRAPH 2016 Courses*, ser. SIGGRAPH ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2897826.2927348>
- [19] Y. Feng, X. Feng, Y. Shang, Y. Jiang, C. Yu, Z. Zong, T. Shao, H. Wu, K. Zhou, C. Jiang, and Y. Yang, “Gaussian splashing: Unified particles for versatile motion synthesis and rendering,” *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 518–529, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:271333936>
- [20] M. Macklin, M. Müller, and N. Chentanez, “Xpbd: position-based simulation of compliant constrained dynamics,” *Proceedings of the 9th International Conference on Motion in Games*, 2016.
- [21] J. Waczyńska, P. Borycki, J. Kaleta, S. a. Tadeja, and P. a. Spurek, “D-miso: Editing dynamic 3d scenes using multi-gaussians soup,” in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 107 865–107 889. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf
- [22] X. Tong, T. Shao, Y. Weng, Y. Yang, and K. Zhou, “As-Rigid-As-Possible Deformation of Gaussian Radiance Fields,” *IEEE Transactions on Visualization & Computer Graphics*, no. 01, pp. 1–13, Mar. 5555. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TVCG.2025.3555404>
- [23] M. S. Floater, “Mean value coordinates,” *Comput. Aided Geom. Des.*, vol. 20, 2003.
- [24] T. Ju, S. Schaefer, and J. D. Warren, “Mean value coordinates for closed triangular meshes,” *ACM SIGGRAPH 2005 Papers*, 2005.
- [25] T. DeRose and M. Meyer, “Harmonic coordinates,” Pixar Animation Studios, Pixar Technical Memo 06-02, 2006. [Online]. Available: <http://graphics.pixar.com/library/HarmonicCoordinates/paper.pdf>
- [26] P. Joshi, M. Meyer, T. DeRose, B. Green, and T. Sanocki, “Harmonic coordinates for character articulation,” *ACM Trans. Graph.*, vol. 26, no. 3, p. 71–es, jul 2007. [Online]. Available: <https://doi.org/10.1145/1276377.1276466>
- [27] Y. Lipman, D. Levin, and D. Cohen-Or, “Green coordinates,” *ACM SIGGRAPH 2008 papers*, 2008.
- [28] S. Li and Y. Pan, “Interactive geometry editing of neural radiance fields,” *ArXiv*, vol. abs/2303.11537, 2023.
- [29] S. J. Garbin, M. Kowalski, V. Estellers, S. Szymanowicz, S. Rezaeifar, J. Shen, M. A. Johnson, and J. Valentin, “Voltemorph: Real-time, controllable and generalizable animation of volumetric representations,” *Computer Graphics Forum*, vol. 43, no. 6, p. e15117, 2024. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.15117>
- [30] W. Zielonka, T. Bagautdinov, S. Saito, M. Zollhöfer, J. Thies, and J. Romero, “Drivable 3d gaussian avatars,” 2023.
- [31] S. Calderon and T. Boubekeur, “Bounding proxies for shape approximation,” *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, vol. 36, no. 5, july 2017.
- [32] P. Dai, J. Xu, W. Xie, X. Liu, H. Wang, and W. Xu, “High-quality surface reconstruction using gaussian surfels,” in *ACM SIGGRAPH 2024 Conference Papers*. Association for Computing Machinery, 2024.
- [33] W. Yifan, N. Aigerman, V. G. Kim, S. Chaudhuri, and O. Sorkine-Hornung, “Neural cages for detail-preserving 3d deformations,” in *CVPR*, 2020.
- [34] T. Ju, S. Schaefer, and J. Warren, “Mean value coordinates for closed triangular meshes,” in *ACM Siggraph 2005 Papers*, 2005, pp. 561–566.
- [35] M. Garland and P. S. Heckbert, “Surface simplification using quadric error metrics,” *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997. [Online]. Available: <https://api.semanticscholar.org/CorpusID:621181>
- [36] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. W. Fitzgibbon, “Kinectfusion: Real-time dense surface mapping and tracking,” *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 127–136, 2011. [Online]. Available: <https://api.semanticscholar.org/CorpusID:11830123>
- [37] Y. Seo, Y. S. Choi, H. S. Son, and Y. Uh, “Flod: Integrating flexible level of detail into 3d gaussian splatting for customizable rendering,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.12894>
- [38] Q.-Y. Zhou, J. Park, and V. Koltun, “Open3D: A modern library for 3D data processing,” *arXiv:1801.09847*, 2018.
- [39] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, “Nerf: Representing scenes as neural radiance fields for view synthesis,” in *ECCV*, 2020.
- [40] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, “Mip-nerf 360: Unbounded anti-aliased neural radiance fields,” *CVPR*, 2022.
- [41] A. Pumarola, E. Corona, G. Pons-Moll, and F. Moreno-Noguer, “D-nerf: Neural radiance fields for dynamic scenes,” *arXiv preprint arXiv:2011.13961*, 2020.
- [42] L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt, “Neural sparse voxel fields,” *NeurIPS*, 2020.

- [43] Blender Online Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Blender Institute, Amsterdam, 2025. [Online]. Available: <http://www.blender.org>
- [44] I. Adobe. (2024) Mixamo: Animate 3d characters for games, film, and more. <https://www.mixamo.com>.
- [45] V. Rong, J. Chen, S. Bahmani, K. N. Kutulakos, and D. B. Lindell, "Gstex: Per-primitive texturing of 2d gaussian splatting for decoupled appearance and geometry modeling," *arXiv preprint arXiv:2409.12954*, 2024.
- [46] A. Haque, M. Tancik, A. Efros, A. Holynski, and A. Kanazawa, "Instruct-nerf2nerf: Editing 3d scenes with instructions," in *Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023*.

VII. BIOGRAPHY SECTION



Jiajun Huang Jiajun Huang is a Ph.D student at the National Centre for Computer Animation (NCCA) of Bournemouth University. He obtained his bachelor's degree from South China Normal University. His research interest covers neural scene representation understanding and editing. Currently, he is conducting research on 3D Gaussian Splatting understanding and animation.



Shuolin Xu Shuolin Xu is a Ph.D student at the National Centre for Computer Animation (NCCA) of Bournemouth University. He obtained his bachelor's degree from Zhengzhou University and his master's degree from Bournemouth University. Currently, he is conducting research on generative models, particularly in the areas of video generation and motion generation.



Hongchuan Yu Hongchuan Yu is a Principal Academic of computer graphics in National Centre for Computer Animation, Bournemouth University, UK. He has published around 110 academic articles in reputable journals and conferences, and regularly served as PC members/referees for international journals and conferences. He is a Member of IEEE (MIEEE) and a fellow of High Education of Academy United Kingdom (FHEA). His research interests include Geometry, GenAI, Graphics, Image, and Video processing.



Tong-Yee Lee Tong-Yee Lee (Senior Member, IEEE) received the Ph.D. degree in computer engineering from Washington State University, Pullman, in 1995. He is currently a chair professor with the Department of Computer Science and Information Engineering, National Cheng-Kung University (NCKU), Tainan, Taiwan. He leads the Computer Graphics Laboratory, NCKU (<http://graphics.csie.ncku.edu.tw>). His current research interests include computer graphics, nonphotorealistic rendering, medical visualization, virtual reality, and media resizing. He is a Senior Member of the IEEE and a Member of the ACM. He also serves on the editorial boards of both the IEEE Transactions on Visualization and Computer Graphics and IEEE Computer Graphics and Applications.

APPENDIX A

PSEUDO-CODE OF CAGE-BUILDING ALGORITHM

The pseudo code of the cage-building algorithm is presented in Algorithm 1 for reference.

Algorithm 1 Cage-Building Algorithm

Require: 3DGS Scene S , config parameters (num_rings n_r , cameras_per_ring n_c , expand factor s)

Ensure: Simplified Cage Mesh C

- ▷ compute expanded bounding sphere ◁
- M = means of Gaussians in S
- m = Mean(M)
- r = Max(L2 distance from points in M to m) * s
- ▷ generate cameras surrounding S , with top and bottom ◁
- A = Generate cameras in n_r evenly spaced rings of n_c cameras each on a sphere of radius r .
- Add cameras to A at the top and bottom poles.
- Orient all cameras in A to look towards the center m .
- ▷ render depth image ◁
- D = Render3DGS(S , A)
- ▷ reconstruct voxel grid ◁
- F_{tsdf} = TSDFIntegration(D , A)
- V_s = ExtractSurfaceVoxel(F_{tsdf})
- D_{clean} = RenderTSDF(F_{tsdf} , A)
- V_i = DepthCarving(D_{clean} , A)
- V = merge voxels in V_s and V_i
- ▷ smooth and simplify ◁
- V_b = MorphoClosing(V)
- C_{raw} = MarchingCube(V_b)
- C_s = BilateralFilter(C_{raw})
- C = EdgeCollapse(C_s)

return C

APPENDIX B

PSEUDO-CODE OF DEFORMATION ALGORITHM

Our deformation algorithm is presented in Algorithm 2. The split function is presented in Algorithm 3. Additionally, it employs EulerToMVC() and MVCToEuler() from cage-based deformation to convert between Euclidean and cage-based coordinates for deformation.

To achieve real-time performance, we perform extensive caching in our implementation. Note that AP_{mvc} and part of T in Algorithm 2 can be precomputed. We also present a simplified implementation of the splitting process that directly operates on the deformed points-represented ellipsoids.

APPENDIX C

DETAILS OF CAGE-BUILDING BASELINE

Inspired by the 3DGS marching cube method proposed by SuGaR [5], our marching cube baseline creates a cage by first converting 3DGS into a binary occupancy voxel grid through opacity thresholding. The process then applies marching cubes to create meshes and simplify them through edge collapse, as proposed by NeRFShop [4].

We transform 3DGS into a binary voxel grid by thresholding each voxel's opacity. A voxel becomes one if its opacity

Algorithm 2 GSDeformer Deformation Algorithm

Require: 3DGS scene S_s , Source Cage C_s , Target Cage C_d , split threshold angle t .

Ensure: Deformed 3DGS scene S_d

ret = Empty3DGS()

for Gaussian i in GS **do**

- ▷ convert Gaussian to points-represented ellipsoids ◁
- AP_s = point set of i using Equation (4)
- ▷ deform points using cage-based deformation ◁
- AP_{mvc} = EulerToMVC(AP_s , C_s)
- AP_d = MVCToEuler(AP_{mvc} , C_d)
- ▷ perform splitting ◁
- AP_{ss} = [AP_s]
- AP_{ds} = [AP_d]
- AP_{ss} , AP_{ds} = Split(i , AP_{ss} , AP_{ds} , t)
- ▷ transform Gaussians ◁
- for** pre/post transform pair s_i , d_i in AP_{ss} and AP_{ds} **do**
- T = transform from s_i to d_i using Equation (6)
- t = c in d_i - c in s_i
- i' = transform i using Equation (7) and (8)
- append i' to ret

return ret

Algorithm 3 Split Function

Require: Gaussian i , source points AP_s , deformed points AP_d , split threshold angle t .

Ensure: split source points AP_s and deformed points AP_d

for axis a in $\{x, y, z\}$ **do**

for deformed ellipsoid d_i in AP_d **do**

- c = d_i 's center
- l, r = endpoints of d_i 's a axis
- α = angle formed by l - c - r
- if** $\alpha < t$ **then**
- ▷ Compute split ellipsoid 1 ◁
- d_l = copy of d_i
- center of d_l = mean(c , l)
- shift rest of d_l points from c to new center
- make c - l d_l 's new a axis
- ▷ Compute split ellipsoid 2 ◁
- d_r = copy of d_i
- center of d_r = mean(c , r)
- shift rest of d_r points from c to new center
- make c - r d_r 's new a axis
- ▷ update ellipsoid sets ◁
- replace d_i in AP_d using two ellipsoids d_l , d_r
- duplicate d_i 's source ellipsoid in AP_s

return AP_s , AP_d

exceeds the threshold and zero otherwise. We calculate voxel opacity $d(\mathbf{v})$ by summing contributions from the K -nearest 3D Gaussians from the voxel center:

$$d(\mathbf{v}) = \sum_g \alpha_g \exp\left(-\frac{1}{2}(\mathbf{v} - \mu_g)^T \Sigma_g^{-1} (\mathbf{v} - \mu_g)\right) \quad (13)$$

For each Gaussian g near voxel center \mathbf{v} , we use its opacity α_g , mean μ_g , and covariance Σ_g . We use a threshold of $1e-6$ and $K=16$ nearest Gaussians.

With the occupancy voxel grid, we smooth it using Bounding Proxy [31], mesh it using marching cubes, and simplify it using edge collapse to create the final cage.

This cage-building method is later extended to work with 2DGS [9] by converting 2DGS to 3DGS: For every flat ellipse in 2DGS, we compute its third axis as the cross product of the first two axes and set its length to $1e-5$, converting 2D ellipses into 3D ellipsoids.