



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Runtime Plane-Based Separation of Deformable Gaussian Splatting Models Toward Interactive Medical Simulation

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA IN-
FORMATICA

Author: **Mario Vallone**

Student ID: 10721136

Advisor: Prof. Marco Gribaudo

Co-advisor: Marco Domenico Buttiglione

Academic Year: 2025-26

Abstract

3D Gaussian Splatting enables real-time rendering of photorealistic reconstructed scenes by representing objects as optimized anisotropic Gaussian primitives. One motivating application is interactive medical simulation, where anatomical models should combine visual realism with physically meaningful manipulation. However, Gaussian Splatting is primarily a visual representation and does not directly provide the physical and topological structure required for interactive deformation or cutting. A common way to introduce deformation is to couple the Gaussian representation to a tetrahedral physical structure, namely a coarse tetrahedral simulation mesh that stores deformable degrees of freedom. The Gaussians provide visual appearance, while their positions and attributes are updated through embedding from the simulated physical structure.

This thesis presents a runtime plane-based separation operation for a physically aware Gaussian Splatting pipeline. The contribution is a complete interaction-to-simulation pipeline that connects Unity, a native C++/CUDA plugin, tetrahedral topology, GPU simulation buffers, and Gaussian embedding data. From the Unity application, a user-defined plane is forwarded to a new native `CutByPlane` API. The native implementation classifies tetrahedral cells with respect to the plane, duplicates vertices shared by cells on opposite sides, remaps the cut-side cells to the duplicated vertices, assigns a new simulation group, rebuilds GPU buffers, and reinitializes the FEM/XPBD simulation data. The Gaussian-to-tetrahedron embedding is then recomputed or repaired so that the rendered splats remain consistent with the separated physical structure.

The resulting method provides a runtime separation mechanism compatible with the existing simulation and rendering data structures. Validation shows that post-cut interaction affects the separated parts independently, demonstrating a change in physical connectivity rather than only visual masking. The validation uses a human body scene as a representative anatomical model, while remaining a technical pipeline validation rather than a clinical evaluation.

Keywords: Gaussian Splatting, deformable simulation, tetrahedral physical structure, tetrahedral embedding, runtime cutting, virtual reality, CUDA

Abstract in lingua italiana

Questa tesi presenta un visualizzatore deformabile basato su Gaussian Splatting con un'operazione interattiva di separazione runtime tramite un piano. Una motivazione applicativa del lavoro è la simulazione medica interattiva, dove modelli anatomici realistici dovrebbero poter essere anche manipolati, deformati e modificati fisicamente. Nei sistemi deformabili basati su Gaussian Splatting, la deformazione può essere introdotta accoppiando le primitive gaussiane a una struttura fisica tetraedrica. Tale struttura è una mesh tetraedrica grossolana composta da vertici e celle volumetriche: la simulazione aggiorna questa rappresentazione fisica, mentre le Gaussiane la seguono tramite embedding.

Il lavoro implementa un flusso completo che collega Unity, il plugin nativo C++/CUDA, la topologia tetraedrica, i buffer GPU della simulazione e l'embedding delle Gaussiane. A partire da un piano definito nell'applicazione Unity, viene invocata una nuova funzione `CutByPlane` nel codice nativo. L'algoritmo classifica le celle tetraedriche rispetto al piano, duplica i vertici condivisi tra i due lati, rimappa le celle del lato separato, assegna un nuovo gruppo di simulazione, ricostruisce i buffer GPU e reinizializza i dati FEM/XPBD. Dopo la modifica della struttura fisica, l'embedding tra Gaussiane e tetraedri viene ricalcolato o corretto per mantenere coerenza tra rendering e simulazione.

Il metodo proposto fornisce una separazione runtime compatibile con le strutture dati di simulazione e rendering esistenti. La validazione mostra che, dopo il taglio, le parti separate possono essere manipolate in modo indipendente, indicando che l'operazione modifica la connettività fisica e non solo la visibilità delle Gaussiane. La scena di validazione utilizza un modello di corpo umano come esempio anatomico rappresentativo, pur rimanendo una validazione tecnica della pipeline e non una valutazione clinica.

Parole chiave: Gaussian Splatting, simulazione deformabile, struttura fisica tetraedrica, embedding tetraedrico, taglio runtime, realtà virtuale, CUDA

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
1.1 Thesis Structure	3
2 State of the Art	5
2.1 Real-Time 3D Scene Representations	5
2.2 Gaussian Splatting Methods	6
2.2.1 Static 3D Gaussian Splatting	6
2.2.2 Dynamic and Deformable Gaussian Splatting	7
2.3 Interactive and Physics-Aware Gaussian Splatting	7
2.4 Deformable Simulation and Embedded Deformation	8
2.4.1 Real-Time Deformable Simulation	8
2.4.2 Tetrahedral Physical Structures and Embedded Deformation	9
2.5 Interactive Cutting and Topology Modification	11
2.6 Comparative Analysis	12
2.7 Research Gap	14
3 Theoretical Foundations	15
3.1 3D Gaussian Representation	15
3.2 Tetrahedral Physical Structure	17
3.3 Embedding of Gaussians in Tetrahedral Physical Structures	19
3.4 Real-Time Deformation with XPBD and FEM Constraints	20
3.5 Plane-Based Classification	22
3.6 Topological Separation by Vertex Duplication	23

3.7	Consistency Between Simulation and Rendering	24
4	Implementation	27
4.1	System Architecture	27
4.2	Base Deformable Gaussian Splatting Pipeline	28
4.3	Desktop Interaction Mode	30
4.4	Cut Plane Selection	31
4.5	Unity-to-Native API Extension	33
4.6	Runtime Plane-Based Separation Algorithm	33
4.6.1	Cell Classification	34
4.6.2	Boundary Vertex Detection	36
4.6.3	Vertex Duplication and Cell Remapping	36
4.6.4	GPU Buffer Rebuild	37
4.6.5	Embedding Recalculation	37
4.6.6	FEM and Rigid State Reinitialization	38
4.7	CUDA Helper Kernels	39
4.8	Plugin Robustness and Diagnostics	39
4.9	Build Workflow	40
5	Experimental Validation	41
5.1	Validation Setup	41
5.2	Functional Validation Method	42
5.3	Cut-Plane Interaction Workflow	44
5.4	Topological Separation Validation	45
5.5	Rendering and Embedding Consistency	47
5.6	Performance Measurement	48
5.7	Robustness Tests	49
5.8	Qualitative Results	50
5.9	Results	51
6	Conclusions and Future Work	53
6.1	Limitations and Future Work	54
6.2	Final Remarks	56
	Bibliography	57
	List of Figures	61

1 | Introduction

Recent progress in neural and point-based scene representations has changed the way three-dimensional content can be reconstructed, rendered, and explored. Among these representations, 3D Gaussian Splatting has become especially relevant for interactive applications because it can render high-quality captured scenes at real-time rates. Instead of representing an object as a triangle mesh with explicit surface connectivity, the scene is modeled as a set of anisotropic Gaussian primitives whose positions, colors, opacities, scales, and orientations are optimized from images. This representation is well suited for photorealistic visualization, but it does not directly provide the topological and physical structure required for interactive editing.

A long-term application domain for this type of technology is interactive medical simulation. In medical training and simulation, anatomical models should not only look realistic, but also respond coherently to physical manipulation, deformation, and topology-changing operations such as separation or cutting. This thesis does not aim to build or clinically validate a medical simulator. Rather, it develops one technical component that is relevant to that direction: a runtime operation that can disconnect a visually realistic deformable model at the level of its physical simulation structure.

Virtual and mixed reality environments introduce an additional requirement: users do not only want to observe reconstructed objects, but also to manipulate them. Interaction requires the rendered representation to respond coherently to external forces, collisions, and editing operations. A physically aware Gaussian Splatting pipeline can address this direction by combining Gaussian rendering with real-time deformable simulation. A central technical idea is to couple the visual Gaussian representation to a tetrahedral physical structure: a coarse tetrahedral simulation mesh composed of vertices and volumetric cells. This physical structure stores the degrees of freedom used by the solver, while the high-resolution Gaussian representation follows it through embedding. The simulation side follows the Position Based Dynamics family [11], with XPBD adding compliance to constraint projection so that stiffness is less tied to the time step and the number of solver iterations [1, 7].

This separation between rendering and simulation is essential for performance. The renderer can display tens of thousands of Gaussian primitives, while the simulation only needs to update a smaller tetrahedral structure. At the same time, the coupling introduces a difficult consistency problem when topology changes. A visual cut is not enough: if the physical structure remains connected, forces can still propagate across the hidden boundary and the object continues to behave as one deformable body. A physically meaningful separation must therefore update the tetrahedral topology and then repair the connection between the updated physical structure and the rendered Gaussians.

This thesis investigates how to add an interactive plane-based separation operation to a deformable Gaussian Splatting viewer. Starting from a VR-GS style Unity and C++/CUDA pipeline [5], the work extends the system with a runtime operation that receives a user-defined plane, classifies tetrahedral cells, duplicates vertices shared across the two sides of the cut, remaps the cut-side cells to the duplicated vertices, rebuilds simulation buffers, and recomputes the Gaussian-to-tetrahedron embedding. The result is a separated deformable object whose parts can be manipulated independently after the cut.

The use of a human body validation scene follows this application perspective. Although the algorithm is not specific to anatomy and could be applied to other deformable Gaussian Splatting models, an anatomical body model is a natural representative case for future medical-simulation workflows. It makes the technical objective concrete: a reconstructed body-like object should preserve photorealistic appearance while supporting physically meaningful deformation and separation.

The motivation for this work comes from the gap between visual reconstruction and interactive physical editing. Gaussian Splatting makes it possible to render captured objects with high visual fidelity, but the Gaussian representation itself does not encode a surface mesh, volumetric cells, material parameters, or physical connectivity. Editing operations that depend on topology therefore need an additional simulation structure.

The tetrahedral physical structure fills this role. It acts as a deformable proxy for the Gaussian model and is the object advanced by the XPBD/FEM simulation: the solver updates the tetrahedral vertices and cells, and the Gaussian primitives follow through embedding weights. This architecture separates visual detail from simulation complexity, but runtime separation makes the coupling more delicate. If vertices are duplicated but the embedding is not rebuilt, splats near the boundary may continue to follow the wrong cells. If invalid Gaussians remain active, visual artifacts can appear near the seam. If simulation buffers are not reinitialized, the solver can operate on inconsistent memory after the topology update.

The main objective of the thesis is to make runtime separation a coherent operation across the whole stack: Unity interaction, managed C# bindings, native C++/CUDA plugin code, tetrahedral topology, GPU simulation buffers, and Gaussian embedding. The operation must be interactive, must modify the physical simulation state, and must keep the rendered Gaussians consistent with the updated physical model.

The thesis makes four main contributions:

- an interaction workflow for defining and previewing a plane-based cut from the Unity application, available through the same native separation path used by the VR and desktop interaction modes;
- a new Unity-to-native `CutByPlane` API that converts the plane into the coordinate convention used by the native plugin and dispatches the request to the active simulation object;
- a runtime separation algorithm that classifies tetrahedral cells, duplicates shared boundary vertices, remaps cut-side cells, assigns a new simulation group, reallocates GPU buffers, and reinitializes the FEM/XPBD state required by the compliant constraint solver;
- an embedding repair step that recomputes Gaussian-to-tetrahedron associations, deactivates invalid seam Gaussians, and prevents visual interpolation from bridging across the newly separated boundary.

1.1. Thesis Structure

The rest of the thesis is organized as follows.

Chapter 2 reviews the state of the art in real-time 3D representations, Gaussian Splatting, deformable simulation, embedded deformation, tetrahedral physical structures, and interactive cutting.

Chapter 3 introduces the theoretical concepts needed to understand the implementation: Gaussian primitives, tetrahedral simulation meshes, barycentric embedding, XPBD/FEM simulation, plane classification, and topology modification by vertex duplication.

Chapter 4 presents the implemented system in detail. It describes the Unity interaction layer, the native plugin API, the runtime separation algorithm, GPU buffer rebuilding, embedding recalculation, and diagnostic logging.

Chapter 5 validates the system through functional tests, qualitative screenshots, robust-

ness checks, performance measurements, and experimental results.

Finally, Chapter 6 summarizes the contributions and discusses limitations and future extensions toward exact remeshing, improved seam handling, and more complete interactive editing workflows.

2 | State of the Art

This chapter reviews the research areas that provide the context for the thesis. The implemented system lies at the intersection of real-time scene representation, Gaussian Splatting, interactive virtual reality, deformable simulation, embedded deformation, and topology-changing editing operations. The purpose of the chapter is not to introduce the mathematical details of these methods; those are discussed in Chapter 3. Instead, the objective is to compare the main families of related work and to identify the specific gap addressed by this thesis.

The central difficulty is that the visual and physical requirements of the system are different. Gaussian Splatting provides high-quality real-time rendering, but a set of Gaussian primitives does not by itself define elastic connectivity, volumetric material, or separable topology. Deformable simulation methods provide physical behavior, but they usually operate on meshes, particles, or volumetric discretizations rather than directly on radiance-field primitives. Interactive cutting makes the combination harder: a cut must change not only what is displayed, but also how the physical model is connected and how the visual representation is embedded in it.

2.1. Real-Time 3D Scene Representations

Interactive applications need a 3D representation that can be reconstructed, rendered, and manipulated at acceptable rates. Traditional computer graphics has relied mainly on triangle meshes. Meshes explicitly store vertices, edges, and faces, and this explicit connectivity makes them well suited to rasterization, collision processing, animation, and many editing operations. Their limitation in the context of captured real-world scenes is that photorealistic asset creation can require difficult geometry reconstruction, texture processing, and material estimation.

Neural Radiance Fields (NeRF) represent a scene as a continuous function mapping spatial position and viewing direction to density and radiance [9]. NeRF demonstrated high-quality novel-view synthesis, but the original formulation requires many neural evaluations

along each camera ray, which makes real-time interaction difficult without additional acceleration structures.

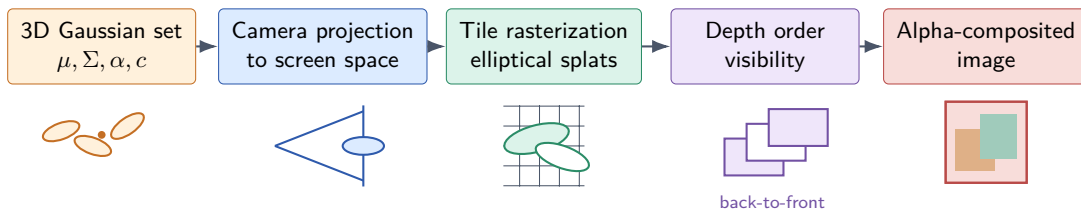
The comparison is important for this thesis. Meshes provide explicit topology but are not always the easiest path to photorealistic reconstruction. NeRFs provide high-quality view synthesis but are expensive to render and do not directly expose editable structure. Point and splat representations render captured detail efficiently, but they also weaken the connection between visual samples and physical topology. The thesis therefore builds on Gaussian Splatting for rendering, while relying on a separate tetrahedral physical structure for simulation and separation.

2.2. Gaussian Splatting Methods

2.2.1. Static 3D Gaussian Splatting

3D Gaussian Splatting represents a scene as a set of anisotropic Gaussian primitives. Each primitive stores a 3D mean, opacity, view-dependent color coefficients, and an anisotropic covariance represented through scale and rotation. During rendering, the Gaussians are projected to screen space and rasterized as elliptical splats. Kerbl et al. introduced an explicit Gaussian representation with density control, anisotropic covariance optimization, and a fast visibility-aware rasterizer [6].

The main strength of 3D Gaussian Splatting is the combination of visual quality and real-time display. In contrast to ray-marched neural fields, a Gaussian scene can be rendered through a rasterization-style pipeline. This makes it attractive for virtual reality, telepresence, immersive visualization, and interactive editing, where latency matters. The resulting rendering path is summarized in Figure 2.1, from explicit Gaussian primitives to the alpha-composited image.



3DGS renders explicit primitives with a rasterization-style pipeline rather than ray marching each pixel.

Figure 2.1: 3D Gaussian Splatting rendering pipeline.

The limitation is that 3D Gaussian Splatting is primarily a visual representation. A

collection of Gaussians stores position, extent, opacity, and appearance, but it does not encode neighboring material elements, rest volumes, elastic constraints, or an explicit surface mesh. This means that 3DGS alone does not provide the information needed for physically meaningful deformation or cutting. For this thesis, 3DGS solves the rendering problem, but not the interactive physical editing problem.

2.2.2. Dynamic and Deformable Gaussian Splatting

Recent work extends Gaussian Splatting from static scenes to dynamic scenes. Deformable 3D Gaussians reconstruct monocular dynamic scenes by learning Gaussians in a canonical space and using a deformation field to map them over time [15]. 4D Gaussian Splatting similarly targets dynamic scene rendering by combining 3D Gaussians with a spatio-temporal representation that predicts time-dependent Gaussian deformations [14].

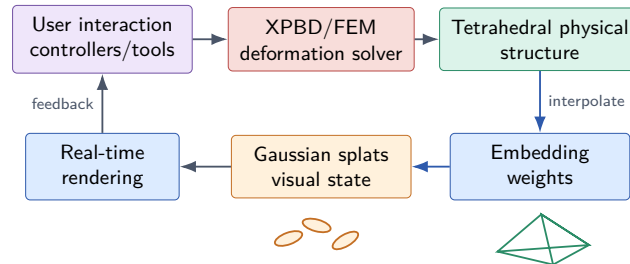
These methods are relevant because they show that Gaussian primitives can be coupled with learned motion and can render time-varying scenes at interactive or real-time rates. They solve a different problem from this thesis, however. Their objective is dynamic reconstruction and novel-view synthesis of observed motion. The motion is learned from data or parameterized by a temporal model, not produced by a user manipulating a physically simulated object in real time. They also do not directly address user-defined topology changes such as cutting, separating, or independently manipulating the two sides of a split object.

Therefore, dynamic Gaussian methods strengthen the case for Gaussians as a powerful visual representation, but they do not remove the need for a physical simulation structure. In this thesis, deformation is not only a rendering animation; it must be driven by an interactive tetrahedral physical structure whose topology can change at runtime.

2.3. Interactive and Physics-Aware Gaussian Splatting

Interactive Gaussian Splatting systems need more than static rendering. Users must be able to select, move, deform, and inspect reconstructed content. VR-GS is the most directly related system to this thesis because it places Gaussian Splatting content in a virtual reality environment and couples it with physical dynamics [5]. The system combines segmented Gaussian assets, deformable body simulation, and a two-level embedding strategy so that users can interact with reconstructed objects rather than only observe them.

The key architectural idea of VR-GS is the separation between the visual representation and the simulation representation. The Gaussian primitives provide appearance, while a tetrahedral physical structure provides the physical degrees of freedom. The visual model follows the simulated tetrahedral structure through embedding. This keeps the simulation cost lower than simulating each Gaussian independently and allows realistic-looking captured objects to respond to user interaction. This separation between Gaussian rendering, physical simulation, and embedding is illustrated in Figure 2.2.



VR-GS separates appearance from physics: Gaussians render the object, while the physical structure carries deformation.

Figure 2.2: VR-GS style architecture with Gaussian rendering and a tetrahedral physical structure.

For this thesis, VR-GS provides the closest starting point: it already connects Gaussian Splatting, VR interaction, and deformable simulation. Its limitation is that deformation does not imply separation. If the tetrahedral physical structure remains connected, forces can still propagate across a visually hidden boundary. A runtime cut therefore requires an additional topology update on the simulation structure and a corresponding update of the Gaussian embedding. This thesis focuses on that missing operation.

2.4. Deformable Simulation and Embedded Deformation

2.4.1. Real-Time Deformable Simulation

Deformable object simulation has been studied for decades in computer graphics, virtual reality, games, and surgical simulation. The literature includes mass-spring systems, continuum-mechanics approaches, finite element methods, meshless methods, and position-based methods. Surgical-simulation surveys emphasize the persistent trade-off between biomechanical realism and interactivity [8, 16]. This trade-off is also present in interactive Gaussian Splatting: the system must remain stable and responsive even if the

physical model is approximate.

Position Based Dynamics (PBD) is widely used in interactive graphics because it projects positions to satisfy constraints rather than relying only on explicit force integration [11]. Extended Position Based Dynamics (XPBD) improves PBD by introducing compliance and reducing the dependence of apparent stiffness on the time step and solver iteration count [7]. Surveys of position-based simulation describe these methods as practical for real-time applications because they are stable, controllable, and suitable for interactive constraints [1]. Within the broader taxonomy in Figure 2.3, PBD and XPBD are therefore the most relevant family for the real-time behavior required by this thesis.

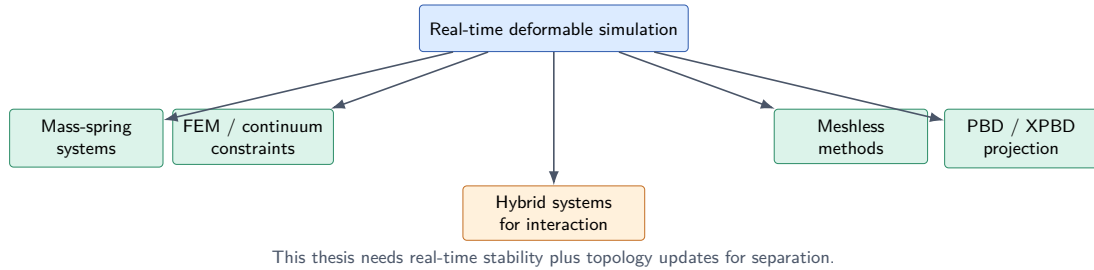


Figure 2.3: Taxonomy of deformable simulation methods.

For this thesis, PBD and XPBD solve the real-time deformation problem, but they do not solve the topology problem by themselves. A solver can deform a connected tetrahedral mesh, but a cutting operation must first modify the mesh connectivity or degrees of freedom so that the solver no longer treats the two sides as a single continuous object.

2.4.2. Tetrahedral Physical Structures and Embedded Deformation

Embedding is a common strategy for separating simulation resolution from visual resolution. A coarse physical structure stores the degrees of freedom, while a more detailed visual model follows it through interpolation or deformation weights. Sumner et al. formalized embedded deformation as a way to manipulate objects embedded in space independently of their specific representation, using a reduced deformation graph to drive detailed geometry [13]. The same broad idea appears in tetrahedral simulation meshes: the coarse physical structure controls motion, while the high-resolution surface, particles, or splats follow.

In the VR-GS setting, this coarse structure is a tetrahedral physical structure. It is a volumetric mesh made of tetrahedral cells that share vertices. The vertices store the

simulated degrees of freedom, while the tetrahedral cells define the local volumetric connectivity used by the deformation solver. This structure is not the rendered object itself. Instead, it is a physical proxy: the solver updates the tetrahedral mesh, and the Gaussian representation is reconstructed from that deformation through embedding weights.

This design is effective because the number of Gaussians can be much larger than the number of simulation vertices. A dense visual representation can therefore follow a much smaller physical discretization. The approach also explains why topology changes are non-trivial. If two tetrahedral cells share a vertex, they share a simulation degree of freedom; forces can propagate through that shared state. A physical cut must modify this connectivity, not only hide or move visual primitives. This embedding relationship between a dense visual representation and a coarser tetrahedral physical structure is shown in Figure 2.4.

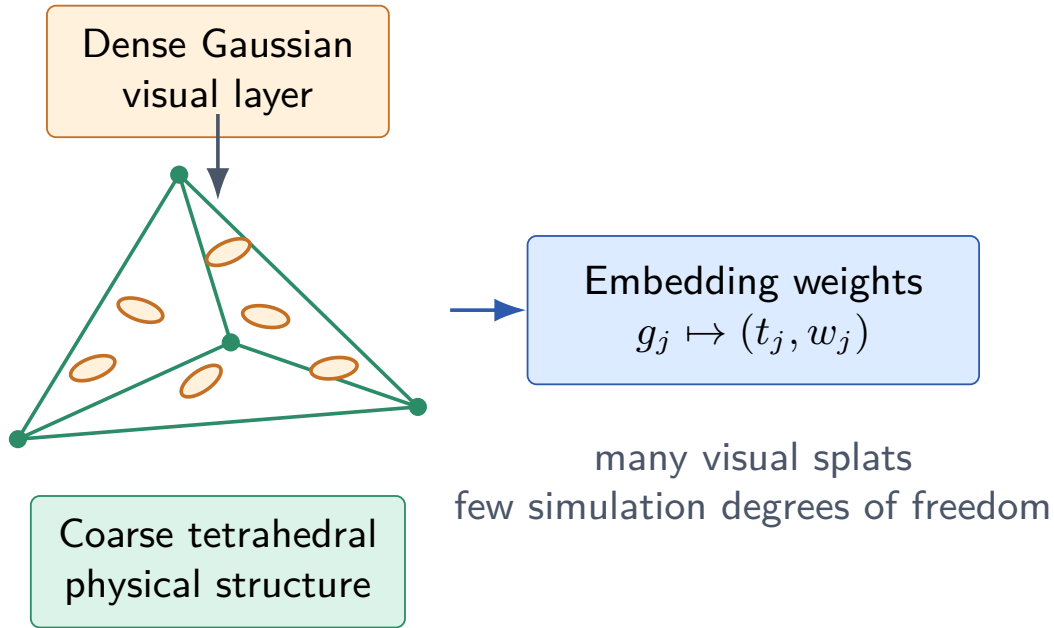


Figure 2.4: High-resolution visual representation embedded in a coarse tetrahedral physical structure.

The limitation of embedding is that it assumes the mapping remains valid. When the simulation topology changes, old embedding references can attach visual primitives to the wrong side of a cut or interpolate across a boundary that should be disconnected. For this reason, runtime separation must update both the tetrahedral topology and the embedding data used by the renderer.

2.5. Interactive Cutting and Topology Modification

Cutting deformable objects is difficult because it changes geometry, topology, simulation state, collision structures, and rendering at the same time. In a mesh-based simulation, a cutting algorithm may need to detect intersections between a tool and the mesh, create new vertices and faces, update volumetric elements, preserve numerical stability, and render newly exposed material. The problem is especially visible in surgical simulation, where the operation must remain interactive and physically plausible.

Early work on soft-tissue cutting investigated interactive cuts through tetrahedral soft-tissue models [2]. Hybrid surgical simulation models also combined real-time deformation, force feedback, and cutting-oriented approximations [4]. Later work introduced more explicit topology-management strategies. Bielser et al. proposed a state machine for real-time cutting of tetrahedral meshes, tracking topological patterns and inserting intersection structures progressively [3]. The virtual node algorithm introduced another important family of methods by allowing material to separate through node and element duplication while preserving well-shaped simulation elements [10]. Sifakis et al. further discussed arbitrary cutting of tetrahedralized deformable objects, emphasizing the difficulty of placing cuts while avoiding poor-quality sliver elements and preserving detailed embedded surfaces [12]. The comparison in Figure 2.5 places these approaches along the practical spectrum from visual masking to exact remeshing.

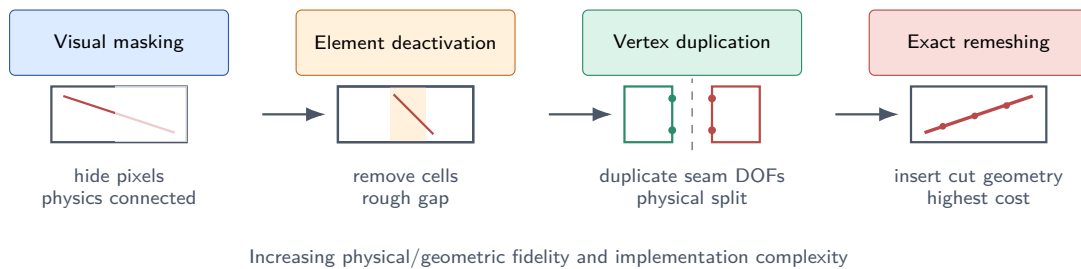


Figure 2.5: Comparison between visual masking, element removal, vertex duplication, and exact remeshing.

These works show that cutting can be approached at different levels of complexity:

- **Visual masking** hides elements or pixels but leaves the physical model connected. It is efficient, but it does not produce physical separation.
- **Element removal or deactivation** changes the simulation model but can leave gaps, rough boundaries, or disconnected visual data.

- **Subdivision and remeshing** can represent the cut surface more accurately, but it increases implementation complexity and may introduce poorly conditioned elements.
- **Virtual-node and duplication methods** preserve simulation element quality by duplicating degrees of freedom or elements, but require careful bookkeeping to keep rendering, collision, and physical state consistent.

The method implemented in this thesis is closest in spirit to the last category, but it is deliberately simpler. It does not compute exact plane-tetrahedron intersections, does not create new tetrahedra, and does not synthesize a new cut surface. Instead, it classifies existing tetrahedral cells with respect to a user-defined plane, duplicates shared boundary vertices, and remaps the cells on one side to the duplicated degrees of freedom. This creates a topological separation compatible with the existing runtime data structures. The trade-off is that the cut follows the resolution of the original tetrahedral physical structure.

2.6. Comparative Analysis

Table 2.1 summarizes the main families of related work discussed in this chapter. The comparison focuses on the role each method can play in this thesis and on the limitation that prevents it from directly solving runtime physical separation for deformable Gaussian Splatting.

Table 2.1: Comparison of related representation, deformation, and cutting approaches.

Method family	Representative works	Contribution and limitation for this thesis
NeRF	NeRF [9]	Introduces high-quality novel-view synthesis from posed images, but the original formulation relies on expensive ray marching and does not expose physical topology that can be separated by interaction.
Static 3DGS	3DGS [6]	Replaces dense neural queries with explicit Gaussian primitives and real-time splatting, but remains a visual representation without volumetric connectivity or a physical separation model.
Dynamic 3DGS	Deformable 3D Gaussians [15]; 4DGS [14]	Extends Gaussian rendering to time-varying scenes and learned deformation, but does not target user-driven topology changes such as runtime cutting or physical separation.
VR-GS	VR-GS [5]	Couples Gaussian rendering with a deformable tetrahedral physical structure and VR interaction, establishing the base system for this thesis. Its editing model supports deformation, not runtime physical separation of the physical structure.
PBD/XPBD	PBD [11]; XPBD [7]	Provides stable and controllable real-time deformation, but assumes that the underlying constraint structure is defined. Cutting requires an additional topology update mechanism.
Embedded physical structures	Embedded deformation [13]	Shows how detailed visual data can be driven by a reduced deformation structure. After a cut, the embedding must be recalculated or visual primitives may still bridge the separated region.
Tetrahedral cutting	State-machine cutting [3]; virtual nodes [10]; arbitrary cutting [12]	Provides principled ways to modify tetrahedral topology and represent cuts, often through subdivision, virtual elements, or embedded surfaces. These methods are more complete than visual masking but add substantial remeshing and robustness complexity for a first runtime extension of VR-GS.
This thesis	Plane-based separation by boundary vertex duplication	Implements a practical runtime separation path that updates topology, GPU buffers, and Gaussian embedding. The tradeoff is an approximate cut boundary without exact remeshing or explicit cut-surface generation.

2.7. Research Gap

The reviewed literature shows that the required components exist separately. Gaussian Splatting provides a high-quality real-time visual representation. Dynamic Gaussian methods extend the representation to learned motion. VR-GS connects Gaussian rendering to a deformable tetrahedral physical structure for interactive VR manipulation. PBD and XPBD provide stable real-time deformation. Embedded deformation and simulation structures provide a way to drive detailed visual representations from lower-dimensional physical structures. Cutting literature provides methods for topology-changing deformation, including remeshing and virtual-node-style separation.

What is missing is the combination needed by this thesis: a runtime, user-triggered physical separation operation for a deformable Gaussian Splatting object. A cut must not be treated only as a rendering effect. It must modify the physical topology and keep the Gaussian embedding consistent with the new simulation state. Otherwise, the object may appear separated while forces or visual interpolation still cross the boundary.

The implemented solution addresses this gap with a practical approximation: plane-based cell classification, boundary vertex duplication, cell remapping, GPU buffer rebuilding, simulation reinitialization, and Gaussian embedding repair. This does not replace exact remeshing or surgical cutting methods. Instead, it provides a first runtime separation pipeline that fits the architecture of a deformable Gaussian Splatting viewer and establishes a basis for future work on exact cut-surface generation, higher-quality remeshing, connected-component extraction, and more complete immersive editing workflows.

3 | Theoretical Foundations

This chapter introduces the theoretical concepts required to understand the runtime cutting and separation pipeline described in Chapter 4. The goal is not to provide a complete treatment of Gaussian Splatting, continuum mechanics, or numerical simulation. Instead, the chapter focuses on the specific ideas used by the implementation: Gaussian primitives as a renderable representation, tetrahedral physical structures as a simulation representation, embedding as the bridge between them, position-based deformation as the real-time solver model, and plane-based topology separation as the cutting approximation.

The implemented system combines two different views of the same object. The rendering view is a set of anisotropic 3D Gaussians, each carrying geometric and appearance information. The physical view is a tetrahedral mesh that stores the degrees of freedom used by the deformable simulation. The visual Gaussians do not move independently. Instead, they are embedded into the tetrahedral mesh and updated from the deformation of the physical structure. This separation between visual detail and physical degrees of freedom is essential for real-time interaction: the renderer can display a dense, high-quality representation, while the solver only needs to update a coarser volumetric structure.

Cutting introduces a further requirement. A cut must not only change what is visible; it must also change how forces propagate through the simulated object. If the visual representation is modified but the tetrahedral mesh remains connected, the object may appear separated while still behaving as a single piece. The theoretical foundation for the implemented solution is therefore a topological operation: shared degrees of freedom along the cut boundary are duplicated so that the two sides can evolve independently.

3.1. 3D Gaussian Representation

3D Gaussian Splatting represents a scene as a collection of volumetric Gaussian primitives. Each primitive contributes color and opacity to the final image when projected onto the camera plane. Unlike a triangle mesh, where geometry is usually represented by vertices and faces, a Gaussian Splatting model stores a large number of smooth, spatially

distributed primitives. These primitives can represent complex appearance with high visual quality and can be rendered in real time with an appropriate rasterization pipeline.

A single Gaussian primitive is centered at a 3D position $\boldsymbol{\mu} \in \mathbb{R}^3$. Its spatial extent is described by an anisotropic covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{3 \times 3}$. In practice, this covariance is often parameterized by a scale vector and a rotation. The scale determines the principal radii of the Gaussian ellipsoid, while the rotation determines its orientation in space. The covariance can be written as:

$$\boldsymbol{\Sigma} = \boldsymbol{R} \boldsymbol{S} \boldsymbol{S}^T \boldsymbol{R}^T, \quad (3.1)$$

where $\boldsymbol{R}$ is a rotation matrix and $\boldsymbol{S}$ is a diagonal matrix built from the scale parameters. The main parameters associated with a single anisotropic Gaussian primitive are illustrated in Figure 3.1.

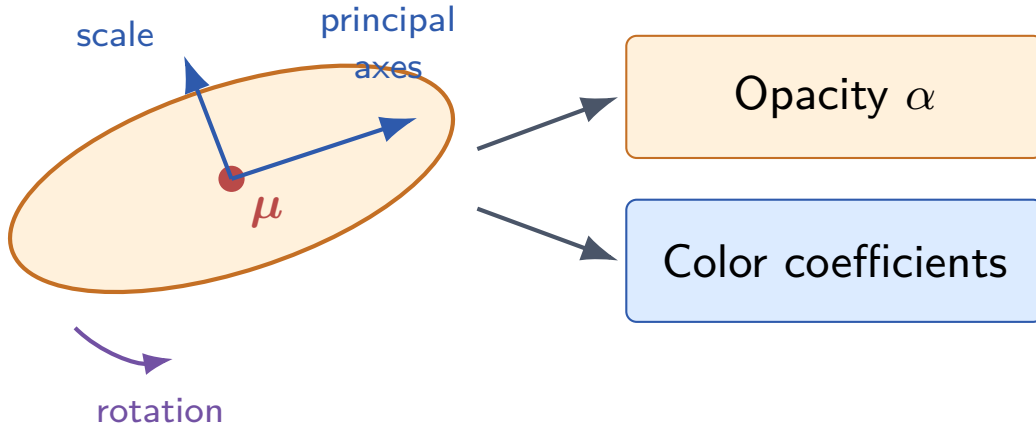


Figure 3.1: Parameters of a 3D Gaussian primitive.

The appearance of a Gaussian is determined by color coefficients and opacity. In the base system, color is represented through spherical harmonics coefficients. This makes the observed color view-dependent: the same Gaussian can contribute different colors depending on the viewing direction. Opacity controls how strongly the Gaussian contributes to the accumulated pixel color and alpha during rendering.

When rendering, each 3D Gaussian is projected onto the image plane. This should be understood as a rendering operation, not as a physical deformation of the Gaussian. From the point of view of the camera, the 3D ellipsoid covers an elliptical region of the screen; this region is the 2D splat footprint shown in Figure 3.2. The rasterizer uses this footprint

to determine which pixels can receive color and opacity contributions from the Gaussian. Because many projected footprints can overlap, the final image is obtained by compositing their contributions in visibility order or by an equivalent rasterization strategy.

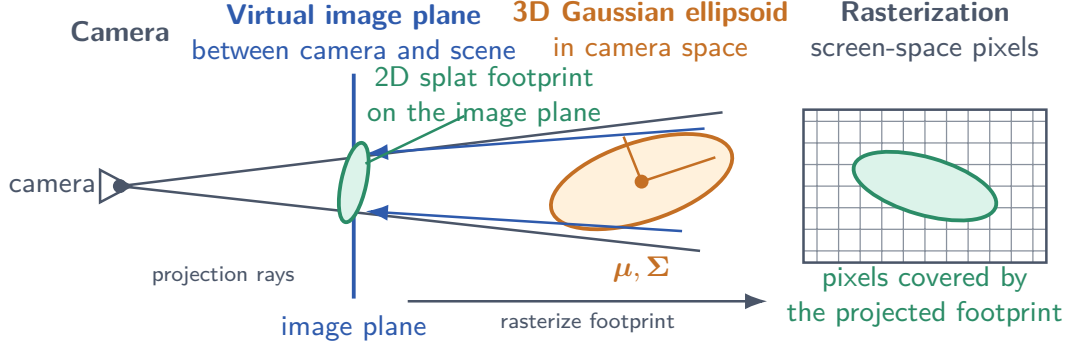


Figure 3.2: Projection of a 3D Gaussian ellipsoid to a 2D splat footprint.

For a deformable simulation, the important point is that Gaussian primitives are not naturally connected to each other. A Gaussian model does not directly encode volumetric connectivity, elastic material properties, or internal constraints. Therefore, if one wants to apply physically meaningful deformation to a Gaussian Splatting object, an additional simulation structure is needed. In this thesis, that structure is a tetrahedral physical structure.

3.2. Tetrahedral Physical Structure

A tetrahedral physical structure can be understood as a volumetric counterpart of a triangle mesh. A triangle mesh describes only the surface of an object, while a tetrahedral mesh fills the interior of the object with small tetrahedral cells. In this thesis, this structure is not rendered directly. It acts as a hidden physical proxy used by the simulation solver: its vertices store the degrees of freedom that move during deformation, while its cells define how the object is connected internally.

More formally, a tetrahedral mesh is a volumetric discretization of a 3D domain. It consists of vertices and tetrahedral cells. Each tetrahedron is defined by four vertex indices. The vertices store positions and simulation state, while the cell connectivity defines which vertices form each volumetric element.

Let a tetrahedral mesh have n vertices. The rest position of vertex i is denoted by $\mathbf{X}_i$, while its current deformed position is denoted by $\mathbf{x}_i$. The rest configuration describes the

undeformed object. The current configuration changes over time as forces and constraints are applied.

A tetrahedron t is defined by four vertex indices:

$$t = (i_0, i_1, i_2, i_3). \quad (3.2)$$

The corresponding rest positions are $\mathbf{X}_{i_0}, \mathbf{X}_{i_1}, \mathbf{X}_{i_2}, \mathbf{X}_{i_3}$, and the current positions are $\mathbf{x}_{i_0}, \mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \mathbf{x}_{i_3}$, as shown in Figure 3.3 for the rest and deformed configurations.

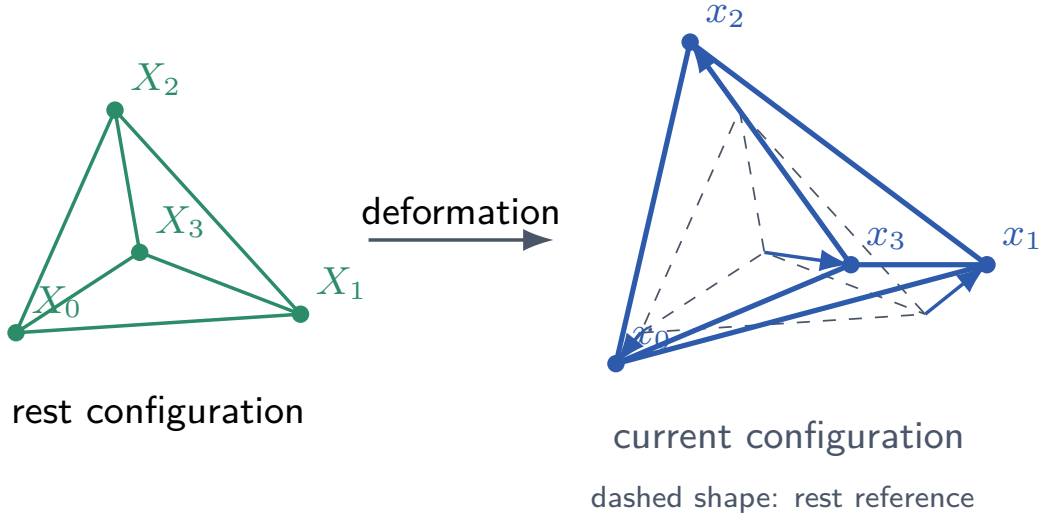


Figure 3.3: Tetrahedral cell with four vertices in rest and deformed configurations.

The tetrahedral mesh provides the connectivity required for physical simulation. Cells share vertices with neighboring cells. This shared-vertex structure creates a deformation graph: when a vertex moves, all cells that reference it are affected. In a continuous object, this connectivity is desired because it allows forces and deformation to propagate through the body. In a cut object, however, connectivity across the cut boundary must be removed.

In the implementation, tetrahedral meshes are loaded from text files containing vertex coordinates and cell indices. These meshes are used as tetrahedral physical structures for the Gaussian Splatting objects. The physical structure can be much coarser than the number of Gaussians, which keeps the simulation cost manageable.

3.3. Embedding of Gaussians in Tetrahedral Physical Structures

The rendering representation and the simulation representation must be connected by an embedding. The embedding defines how a Gaussian primitive follows the motion of the tetrahedral physical structure. Without this bridge, the solver could deform the mesh, but the rendered Gaussian model would not move with it.

Conceptually, each Gaussian is associated with a tetrahedral element in the rest configuration. Its position can be expressed with barycentric coordinates inside that tetrahedron. For a point $\mathbf{p}$ embedded in a tetrahedron with rest vertices $\mathbf{X}_0, \mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3$, barycentric weights w_0, w_1, w_2, w_3 satisfy:

$$\mathbf{p} = w_0 \mathbf{X}_0 + w_1 \mathbf{X}_1 + w_2 \mathbf{X}_2 + w_3 \mathbf{X}_3, \quad (3.3)$$

with

$$w_0 + w_1 + w_2 + w_3 = 1. \quad (3.4)$$

When the tetrahedron deforms, the embedded point is reconstructed from the current vertex positions, completing the barycentric embedding depicted in Figure 3.4:

$$\mathbf{p}' = w_0 \mathbf{x}_0 + w_1 \mathbf{x}_1 + w_2 \mathbf{x}_2 + w_3 \mathbf{x}_3. \quad (3.5)$$

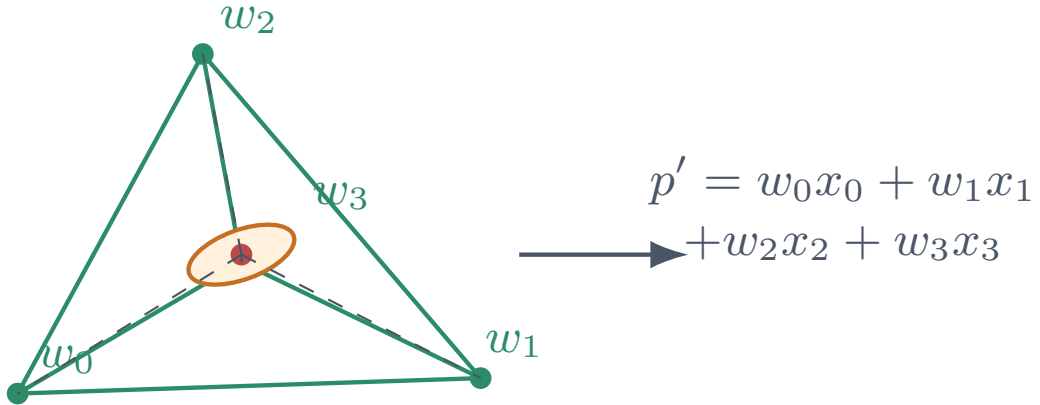


Figure 3.4: Gaussian primitive embedded into a tetrahedral physical structure.

The base system uses a more elaborate embedding strategy to update not only Gaussian positions, but also orientation and scale-related attributes. However, the central idea remains the same: the Gaussian representation is derived from the tetrahedral deformation state. This is why the embedding must be recomputed after a runtime topology change. If a cut duplicates vertices and remaps cells, old embedding references can become invalid or can accidentally blend across the new separation boundary.

An invalid embedding occurs when a Gaussian can no longer be associated with a valid tetrahedral element. In that case, the Gaussian cannot be reliably updated from the physical structure. The implemented system handles such cases by deactivating the Gaussian through its opacity. Another problematic case occurs near the seam: a Gaussian may remain valid, but its interpolation references may span both sides of the cut. Such a Gaussian can visually bridge the two sides even after the physical mesh has been disconnected. The implementation addresses this by reassigning seam embeddings where possible and deactivating them when a valid assignment cannot be maintained.

3.4. Real-Time Deformation with XPBD and FEM Constraints

The simulation layer is responsible for updating the tetrahedral physical structure under external forces and constraints. The implementation uses a real-time position-based approach with FEM-inspired constraints, following the line of PBD and XPBD methods used for stable interactive simulation [1, 7, 11]. This combination is appropriate for interactive systems because it favors stability and controllable behavior under finite time steps.

In a classical force-based simulation, accelerations are integrated from forces, and constraints are often enforced through equations of motion. Position Based Dynamics instead works primarily by predicting positions and then projecting them to satisfy constraints [11]. Extended Position Based Dynamics, or XPBD, improves this framework by introducing compliance, making constraint stiffness less dependent on the time step and number of solver iterations [7].

At a high level, each simulation step follows the sequence summarized in Figure 3.5:

1. Apply external effects such as gravity, damping, and controller forces.
2. Predict new vertex positions.
3. Iteratively solve deformation and collision constraints.

4. Apply the position corrections.
5. Update velocities from the change in position.
6. Update the embedded Gaussian representation.

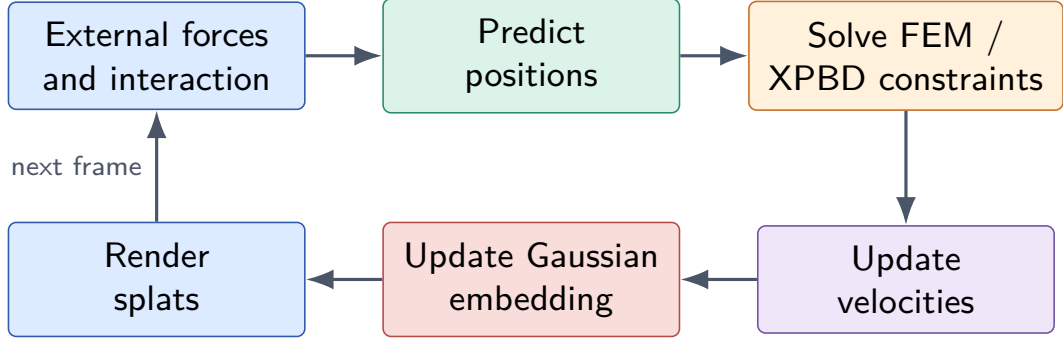


Figure 3.5: XPBD/FEM update loop used by the tetrahedral physical simulation.

For each vertex i , the solver stores a mass m_i and inverse mass $w_i = 1/m_i$. Vertices with zero inverse mass are fixed or constrained by boundary conditions. In each time step, a predicted position can be written in simplified form as:

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i + \Delta t \mathbf{v}_i + \Delta t^2 w_i \mathbf{f}_i, \quad (3.6)$$

where $\mathbf{v}_i$ is velocity, $\mathbf{f}_i$ is force, and Δt is the time step.

Constraints are then solved iteratively. A constraint is represented by a function $C(\mathbf{x})$ that should be zero when satisfied. Position corrections are computed from the constraint value, the constraint gradient, and the inverse masses of the affected vertices. In XPBD, compliance introduces a controlled softness to constraints, which is important for elastic materials and interactive manipulation.

For tetrahedral soft bodies, FEM-inspired constraints depend on the deformation of each tetrahedral cell from its rest configuration to its current configuration. Rest-shape quantities such as inverse basis matrices and rest volumes are computed during initialization. When the topology is changed by a cut, these quantities must be recomputed because the cell index array and vertex arrays may have changed.

The implementation also supports interaction through controller or desktop-hand forces. When a trigger is pressed, vertices within a configurable controller radius are selected. Their motion is then influenced by the controller position, linear velocity, and angular

velocity. This mechanism is independent of whether the input comes from XR controllers or from the desktop interaction mode.

3.5. Plane-Based Classification

The cutting operation is driven by a plane. A plane in 3D can be represented by a point $\mathbf{p}$ on the plane and a unit normal $\mathbf{n}$. Any point $\mathbf{q}$ can be classified relative to the plane using the signed value:

$$d(\mathbf{q}) = (\mathbf{q} - \mathbf{p}) \cdot \mathbf{n}. \quad (3.7)$$

If $d(\mathbf{q}) > 0$, the point lies on the positive side of the plane. If $d(\mathbf{q}) < 0$, it lies on the negative side. If $d(\mathbf{q}) = 0$, it lies on the plane.

In the implemented cut, classification is performed at the level of tetrahedral cells rather than individual vertices. Each tetrahedral cell is one volumetric element of the physical structure introduced in Section 3.2, and is defined by four vertices. During simulation, these vertices have current deformed positions $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$. The representative point used for classification is the cell centroid:

$$\mathbf{c} = \frac{1}{4} (\mathbf{x}_0 + \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3). \quad (3.8)$$

The cell is assigned to one side of the cut based on the sign of $d(\mathbf{c})$. This classification produces two sets of cells. One set is treated as the kept side; the other is treated as the cut side. The keep-side flag selected by the user determines whether the positive or negative side is considered the reference side, following the signed cell-centroid classification shown in Figure 3.6.

$$d(c) = (c - p) \cdot n$$

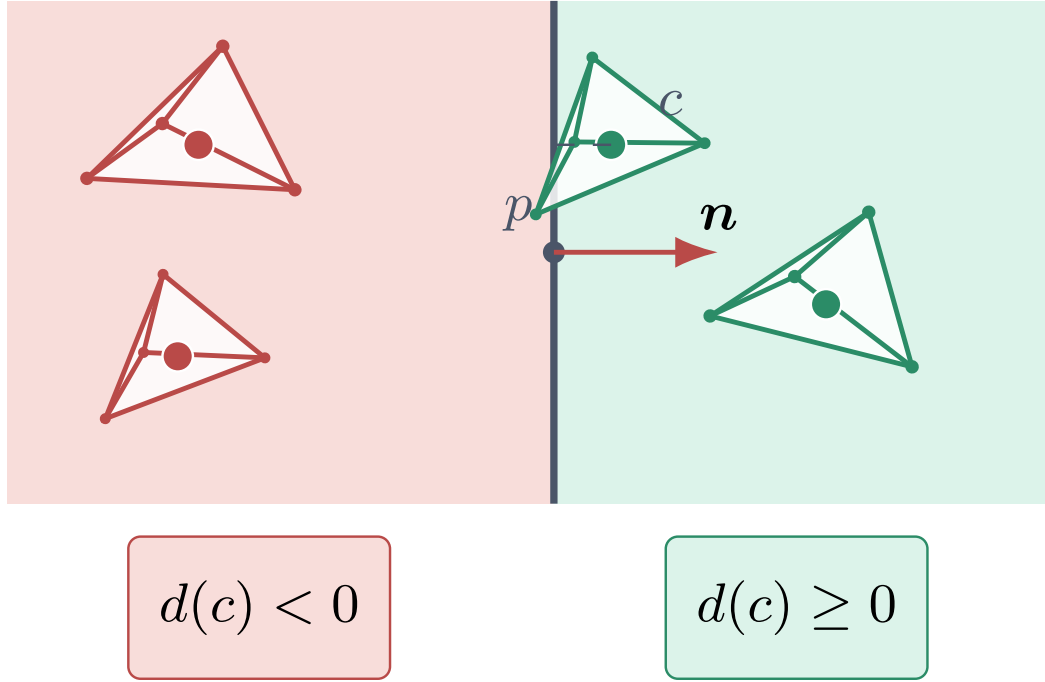


Figure 3.6: Plane classification of tetrahedral cell centroids. Wireframe shapes represent tetrahedral cells schematically, and colored dots mark the centroids used for classification.

This method is efficient and simple to implement, but it is approximate. A tetrahedron that is geometrically intersected by the plane is not split. It is assigned entirely to one side based on its centroid. This approximation avoids the need for dynamic tetrahedral remeshing, which is significantly more complex and would require generating new vertices, faces, cells, and cut-surface geometry.

3.6. Topological Separation by Vertex Duplication

The core theoretical idea behind the implemented separation is that physical connectivity is encoded by shared vertices. If two tetrahedral cells share a vertex, they share a degree of freedom in the simulation. A force applied to one cell can influence the shared vertex, and therefore can affect the other cell. To separate two regions physically, shared degrees of freedom across the boundary must be removed.

After the plane classification, the tetrahedral cells are divided into two groups. Some

vertices are used only by cells in one group; these vertices do not need to change. Other vertices are referenced by at least one cell in each group. These shared boundary vertices are the connection that must be broken, and Figure 3.7 illustrates how duplicating them removes shared simulation degrees of freedom across the cut.

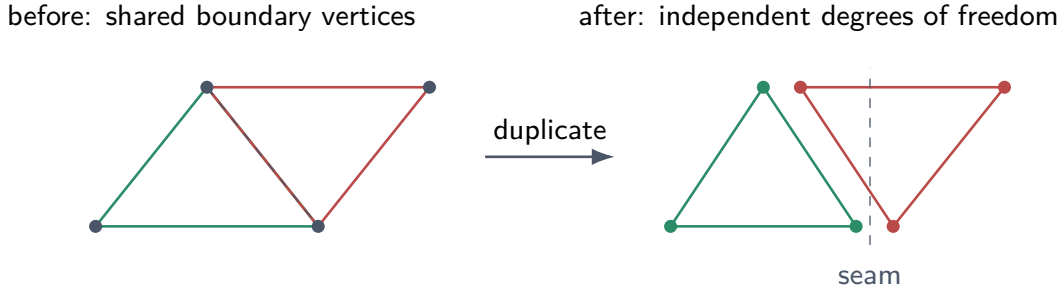


Figure 3.7: Topological separation through boundary vertex duplication.

For each shared boundary vertex v , a duplicate vertex v' is created. The duplicate initially has the same rest position, current position, and velocity as the original:

$$\mathbf{X}_{v'} = \mathbf{X}_v, \quad \mathbf{x}_{v'} = \mathbf{x}_v, \quad \mathbf{v}_{v'} = \mathbf{v}_v. \quad (3.9)$$

The cell connectivity is then updated. Cells on one side of the cut continue to reference the original vertex v . Cells on the other side are remapped to reference the duplicate v' . Geometrically, the object has not yet opened a visible gap at the moment of cutting, because the original and duplicate vertices start at the same position. Topologically, however, the two sides no longer share the same simulation degree of freedom. Subsequent forces can move them apart.

This operation is less general than full remeshing, but it is sufficient to create a physical separation along a boundary induced by the plane classification. It also preserves most of the original tetrahedral structure, which reduces the amount of data that must be rebuilt.

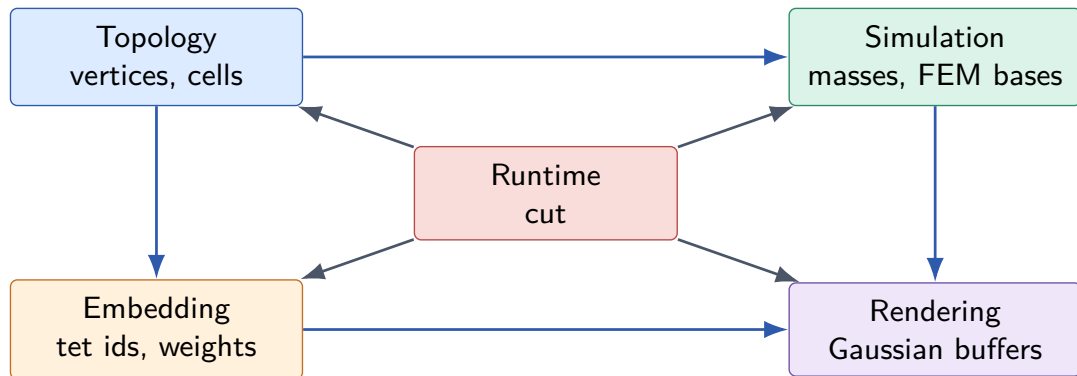
3.7. Consistency Between Simulation and Rendering

A runtime cut affects several layers of the system. Updating only the tetrahedral cell indices is not enough. The solver also depends on masses, inverse masses, rest volumes, FEM basis matrices, temporary correction buffers, selected-vertex buffers, and rigid group data. The renderer depends on Gaussian positions, rotations, scales, opacity values, and embedding references. These data structures must remain consistent after the cut.

The main consistency requirements, summarized by the dependency view in Figure 3.8, are:

- every cell index must reference a valid vertex;
- every simulation buffer must match the current vertex or cell count;
- FEM quantities must be recomputed from the updated rest configuration;
- every active Gaussian must have a valid embedding;
- embedding references should not bridge across separated regions;
- deactivated Gaussians should not contribute to the final image.

If these requirements are not satisfied, the system can produce visual artifacts, unstable simulation, invalid memory access, or crashes. This is why the implementation rebuilds multiple buffers after the topology update rather than trying to patch only the cell index array.



All dependent buffers must agree after topology changes.

Figure 3.8: Data dependencies that must remain consistent after a cut.

4 | Implementation

This chapter describes the implementation of the runtime cutting and separation pipeline developed in this thesis. The work extends an existing deformable Gaussian Splatting viewer based on the VR-GS architecture [5] with an interactive operation that separates a deformable object according to a user-defined plane. The implementation combines Unity interaction logic, a native C++/CUDA plugin, a tetrahedral physical structure, and Gaussian-to-tetrahedron embedding data.

The central objective is to make a cut affect both the visual representation and the physical representation of the object. A simple visual mask would be insufficient: hiding Gaussians on one side of a plane can create the appearance of a cut, but the underlying simulation mesh would remain connected. As a consequence, interaction forces would still propagate across the hidden seam and the two sides of the object would not move independently. For this reason, the implemented method modifies the simulation topology by duplicating vertices that are shared by tetrahedral cells on opposite sides of the cut plane. After this topological update, the simulation state and Gaussian embedding data are rebuilt so that rendering and deformation remain consistent.

The implementation can be divided into five main parts. First, a desktop interaction mode was added to support development and testing without requiring a VR headset. Second, the Unity frontend was extended with a cut-plane selection interface. Third, a new native API call, `CutByPlane`, was exposed from the C++/CUDA plugin to Unity. Fourth, the native simulation code was extended with a runtime plane-based separation algorithm. Finally, the plugin build and diagnostic path was improved to make native failures easier to identify during development.

4.1. System Architecture

The project is composed of two main software layers: a Unity application and a native C++/CUDA plugin. Unity is responsible for scene configuration, user input, camera setup, controller poses, and display integration. The native plugin owns the computa-

tionally expensive parts of the pipeline: Gaussian Splatting rasterization, soft-body simulation, collision handling, and the update of the Gaussian primitives from the deforming tetrahedral physical structure.

The Unity project is contained in `GaussianSplattingVRViewer`. The native plugin is contained in `UnityPlugin`. The compiled native library is exported as `gaussiansplatting.dll` and copied into the Unity plugin folder. At runtime, Unity calls native functions through C# `DllImport` bindings. Rendering is synchronized through Unity plugin events: Unity prepares the camera and draw parameters, then issues a native render event that executes the CUDA update and rasterization path. The resulting runtime architecture and division of responsibilities are shown in Figure 4.1.

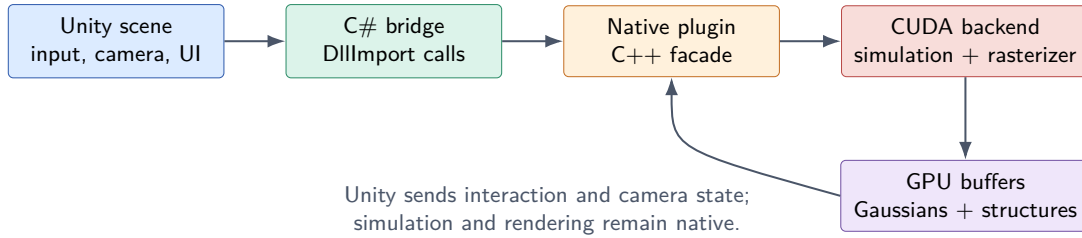


Figure 4.1: Global architecture of the Unity frontend, native plugin, CUDA simulation, and Gaussian renderer.

The main Unity bridge is implemented in `GaussianSplatting.cs`. This component exposes inspector parameters for the model path, camera, simulation settings, lighting settings, and experimental options. It also defines the native function bindings used by the scene. Important calls include `LoadModel`, `SetSimParams`, `SetController`, `SetDrawParameters`, `SetLightingParameters`, `Save`, and the added `CutByPlane`.

On the native side, `plugin.cpp` exposes a C-style API to Unity. `PluginAPI.cpp` provides a higher-level facade around the `GaussianSplatting` simulation object. The core simulation and rendering orchestration is implemented in `GaussianSplatting.cu`. Lower-level CUDA kernels and helper functions are implemented in `CudaKernels.cu` and in the simulation kernel files used by the base system.

4.2. Base Deformable Gaussian Splatting Pipeline

The base pipeline loads a scene as a sequence of segmented Gaussian Splatting objects. Each object is represented by a numbered pair of files. The Gaussian attributes are stored in `i_point_cloud.ply`; the optional tetrahedral physical structure is stored in

`i_tetgen.txt`. If an object has no tetrahedral mesh, it is treated as a static background object. If a tetrahedral mesh is present, the object can participate in the deformable simulation.

The `.ply` files contain the Gaussian position, opacity, scale, rotation, and color coefficients. The `.txt` files define the tetrahedral physical structure with a list of vertices and tetrahedral cells. During loading, all objects are transformed according to the global transform and per-object material parameters configured in Unity. These parameters include density, Young’s modulus, Poisson’s ratio, scale, rigid-body flag, and local rotation; the complete loading path from files to GPU buffers is summarized in Figure 4.2.

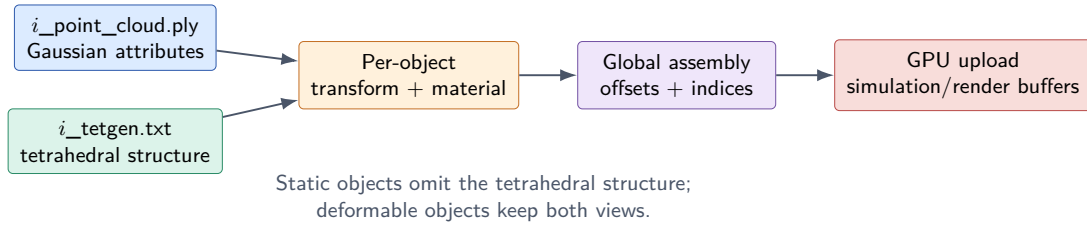


Figure 4.2: Loading pipeline for segmented Gaussian objects and tetrahedral physical structures.

After loading the data, the native code allocates GPU buffers for Gaussian attributes, mesh indices, simulation state, collision data, interpolation data, lighting buffers, and controller state. The implementation stores the tetrahedral mesh using vertex arrays and index arrays for edges, faces, and cells. Simulation state includes rest positions, current positions, velocities, masses, inverse masses, FEM basis matrices, rest volumes, material coefficients, and rigid group data.

The visual representation is connected to the simulation representation through an embedding step. Each Gaussian is associated with one or more tetrahedral elements and interpolation weights. During the simulation update, the tetrahedral physical structure deforms under external forces and constraints. The Gaussian positions, scales, and rotations are then updated from the deformed physical structure through the embedding data. This makes the rendered Gaussian object follow the physical deformation of the simulation mesh.

The per-frame native update follows three broad phases. First, controller positions and velocities are applied as external interaction inputs. Second, the XPBD/FEM simulation loop advances the tetrahedral physical structure. Third, the Gaussian attributes are interpolated from the deformed physical structure and passed to the CUDA rasterizer.

If shadow mapping is enabled, a shadow-map rendering pass is executed before the final camera render pass. This base per-frame update flow, before the runtime cutting extension is applied, is shown in Figure 4.3.

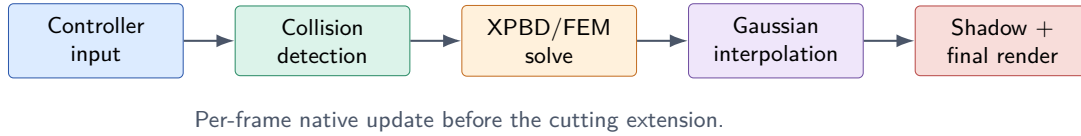


Figure 4.3: Base per-frame update pipeline before the cutting extension.

4.3. Desktop Interaction Mode

The original interaction path is based on XR devices. For development, testing, and debugging, it is useful to run the scene without a headset. This motivated the addition of a desktop interaction mode. The desktop mode keeps the same native simulation and rendering backend, but replaces the XR controller input with a mouse and keyboard interface.

The script `InputForce.cs` now detects whether the `GaussianSplatting` component is configured for XR. If the scene is not running in XR mode, it attaches `DesktopInputForce` to the same `GameObject`, binds the relevant Gaussian Splatting object and hand transforms, transfers the controller radius setting, and disables the XR input script. This allows the same scene to be used both as a headset-driven scene and as a desktop test scene.

The `DesktopInputForce.cs` script creates a desktop camera rig and two virtual hand transforms. The camera can be moved with keyboard input and rotated with the mouse. The virtual hand can be positioned along a camera ray, so that mouse interaction can emulate controller placement. Trigger states are mapped to keyboard and mouse input and are forwarded to the same native `SetController` function used by the XR path. As a result, the native simulation code does not need to distinguish between a real controller and a desktop-controlled virtual hand. The desktop-mode runtime view used to inspect and interact with the loaded deformable Gaussian object is shown in Figure 4.4.

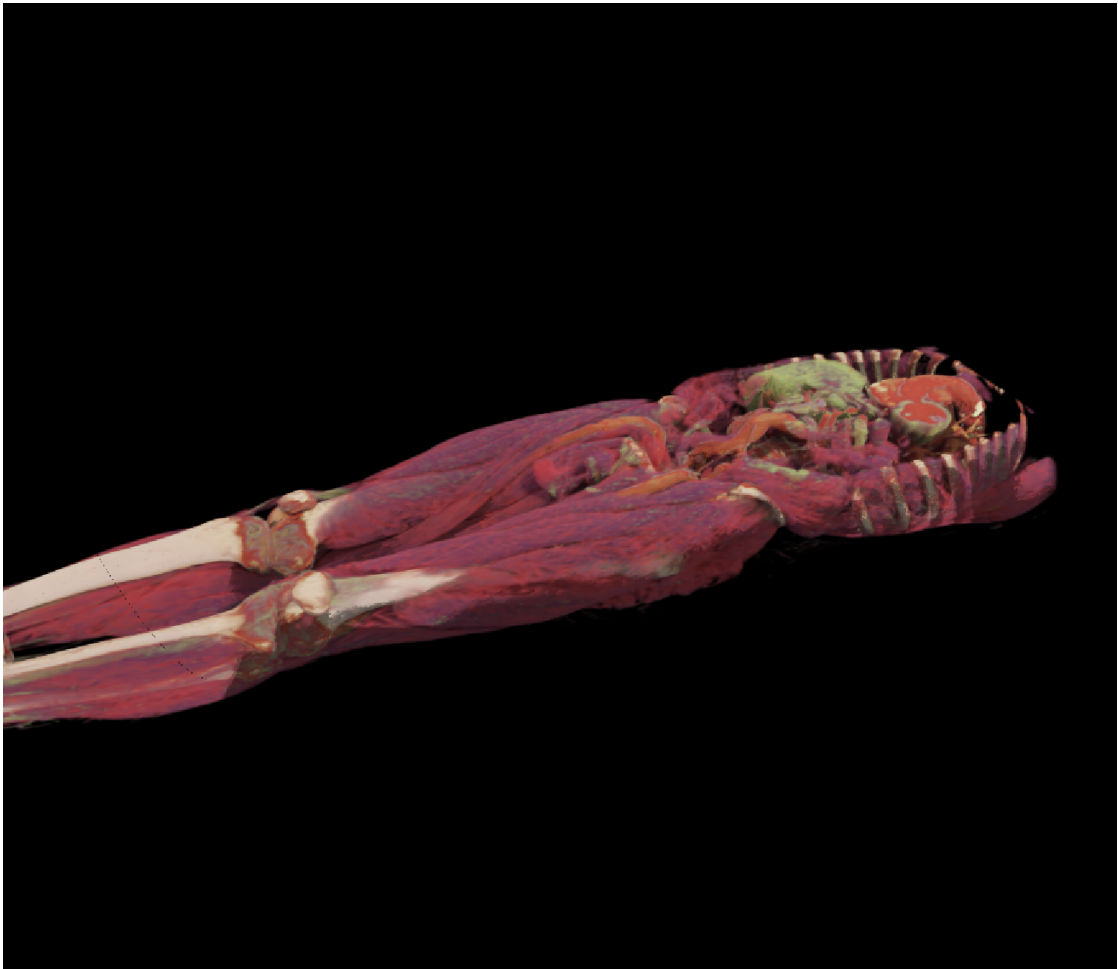


Figure 4.4: Desktop-mode runtime view of the loaded deformable Gaussian object.

The desktop interface also exposes shortcuts for changing the controller radius and saving the current deformed state. The controller radius is important because it determines the region of mesh vertices selected when an interaction trigger is pressed. This parameter is forwarded to the native code together with the controller pose and trigger state.

4.4. Cut Plane Selection

The cutting interaction is implemented as a plane selection tool. A plane is defined by a point and a normal vector. In desktop mode, the user toggles the cut-selection state with a key. While this state is active, the plane point is positioned along the camera ray under the mouse cursor. The plane normal is controlled by a rotation state initialized from the camera direction and adjusted through keyboard input.

Two visual aids are created for the plane. The first is a thin cube in the Unity scene, used

as a translucent preview of the cutting plane. The second is a GUI overlay that draws the plane outline and the normal direction in screen space. These previews are not used by the native simulation; they are only interaction feedback. They make it possible to place and rotate the plane before committing the cut, as shown by the cut-plane preview in Figure 4.5.

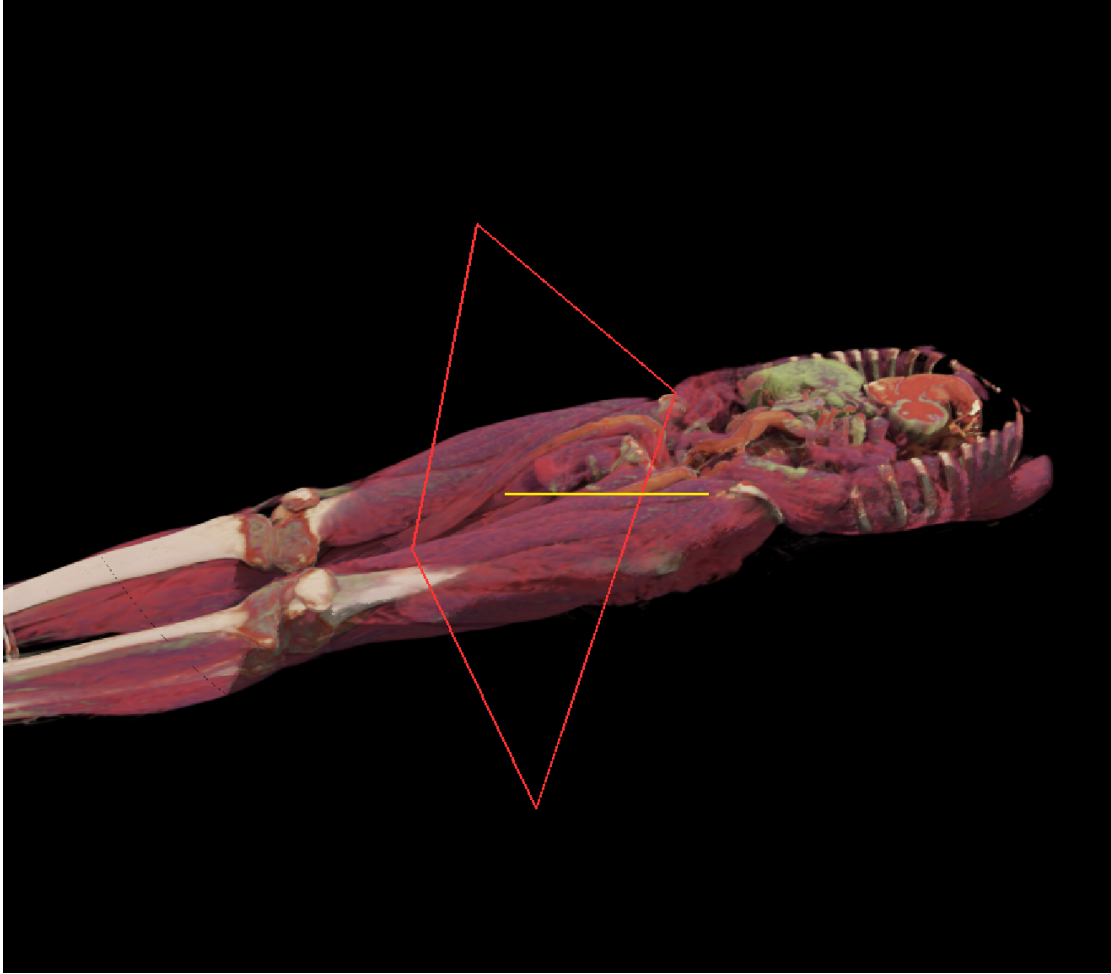


Figure 4.5: Cut-plane preview shown before applying the separation operation. The red outline shows the selected plane intersecting the deformable object.

When the user clicks while cut-selection mode is active, the script calls the public cut method. A modifier key selects which side of the plane is kept as the positive side. The native method returns an integer indicating how many seam Gaussians were reassigned or deactivated by the cut operation. This value is logged in Unity for debugging.

The plane is defined in Unity world coordinates, but the native plugin stores the simulation in the local coordinate system used by the Gaussian Splatting object. Therefore, the C# bridge converts the plane point with `transform.InverseTransformPoint` and the plane

normal with `transform.InverseTransformDirection`. The conversion also accounts for the coordinate-system convention selected by the `isZUp` flag. This is necessary because the plugin expects coordinates in the same convention used by the simulation buffers.

4.5. Unity-to-Native API Extension

The cutting operation is exposed through a new native function named `CutByPlane`. The C# binding is added to the internal native interface class in `GaussianSplatting.cs`. The public method `GaussianSplatting.CutByPlane` performs the coordinate conversion and forwards three values to the plugin: the plane point, the plane normal, and a flag that selects the side to keep.

On the native side, `plugin.h` declares the exported function and `plugin.cpp` implements it. The function checks that the plugin API object exists, forwards the request to the `PluginAPI` facade, logs the result, and catches exceptions. The facade method then calls the `CutByPlane` implementation in the `GaussianSplatting` class, following the call chain shown in Figure 4.6.

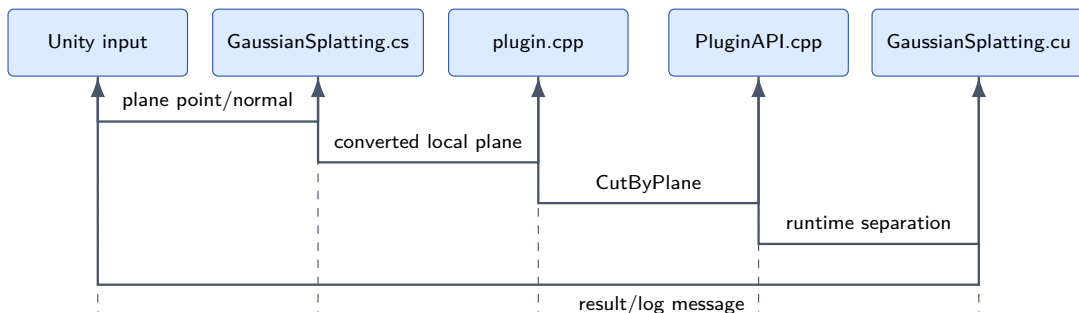


Figure 4.6: API flow from Unity cut-plane interaction to native simulation update.

This layered design follows the existing architecture of the project. Unity does not directly manipulate GPU buffers or tetrahedral topology. It only sends a geometric cutting command. The native plugin is responsible for interpreting that command in the current simulation state and updating all affected data structures.

4.6. Runtime Plane-Based Separation Algorithm

The core implementation is the method `GaussianSplatting::CutByPlane`. This method modifies the tetrahedral simulation topology at runtime. The implemented operation is an approximate topological separation, not a full remeshing operation. It does not compute

exact intersection polygons between the plane and the tetrahedra. Instead, it classifies tetrahedral cells by their center point and duplicates vertices that are shared by cells on both sides of the plane.

The algorithm starts by validating the input. If the plane point, plane normal, Gaussian buffers, or opacity buffers are invalid, the function returns without modifying the scene. The normal is normalized to avoid scale-dependent signed distance values. If the scene has no tetrahedral cells, the implementation falls back to a simpler Gaussian opacity cut described in Section 4.7. When tetrahedral data is available, the full separation path is used.

The native method first copies the relevant simulation state from GPU memory to CPU memory. This includes current vertex positions, rest positions, velocities, vertex groups, rigid groups, and tetrahedral cell indices. The current implementation performs the topology update on the CPU because it requires dynamic arrays, vertex duplication, index remapping, and several temporary data structures. After the topology is rebuilt, the updated arrays are copied back to the GPU.

Algorithm 4.1 Runtime plane-based separation

- 1: Receive plane point, plane normal, and keep-side flag
 - 2: Normalize the plane normal
 - 3: Copy tetrahedral state from GPU to CPU
 - 4: **for** each tetrahedral cell **do**
 - 5: Compute the cell center
 - 6: Classify the center by signed distance to the plane
 - 7: **end for**
 - 8: Detect vertices referenced by cells on both sides
 - 9: Duplicate shared boundary vertices
 - 10: Remap cut-side cells to use duplicated vertices
 - 11: Upload updated topology and simulation state to GPU
 - 12: Reinitialize simulation buffers
 - 13: Recompute Gaussian-to-tetrahedron embedding
 - 14: Reassign or deactivate seam Gaussians
 - 15: Reinitialize FEM bases, inverse masses, and rigid data
-

4.6.1. Cell Classification

Each tetrahedral cell is classified using the signed distance of its centroid to the cutting plane. For a plane point $\mathbf{p}$, normal $\mathbf{n}$, and cell centroid $\mathbf{c}$, the signed distance value is

computed as:

$$s = (\mathbf{c} - \mathbf{p}) \cdot \mathbf{n}. \quad (4.1)$$

If the keep-positive flag is enabled, cells with $s \geq 0$ are marked as the kept side and cells with $s < 0$ are marked as the cut side. If the flag is not enabled, the condition is reversed. The implementation counts the number of cells on each side. If all cells are classified on the same side, the function returns without modifying the topology because no separation boundary exists. This centroid-based classification is visualized in Figure 4.7.

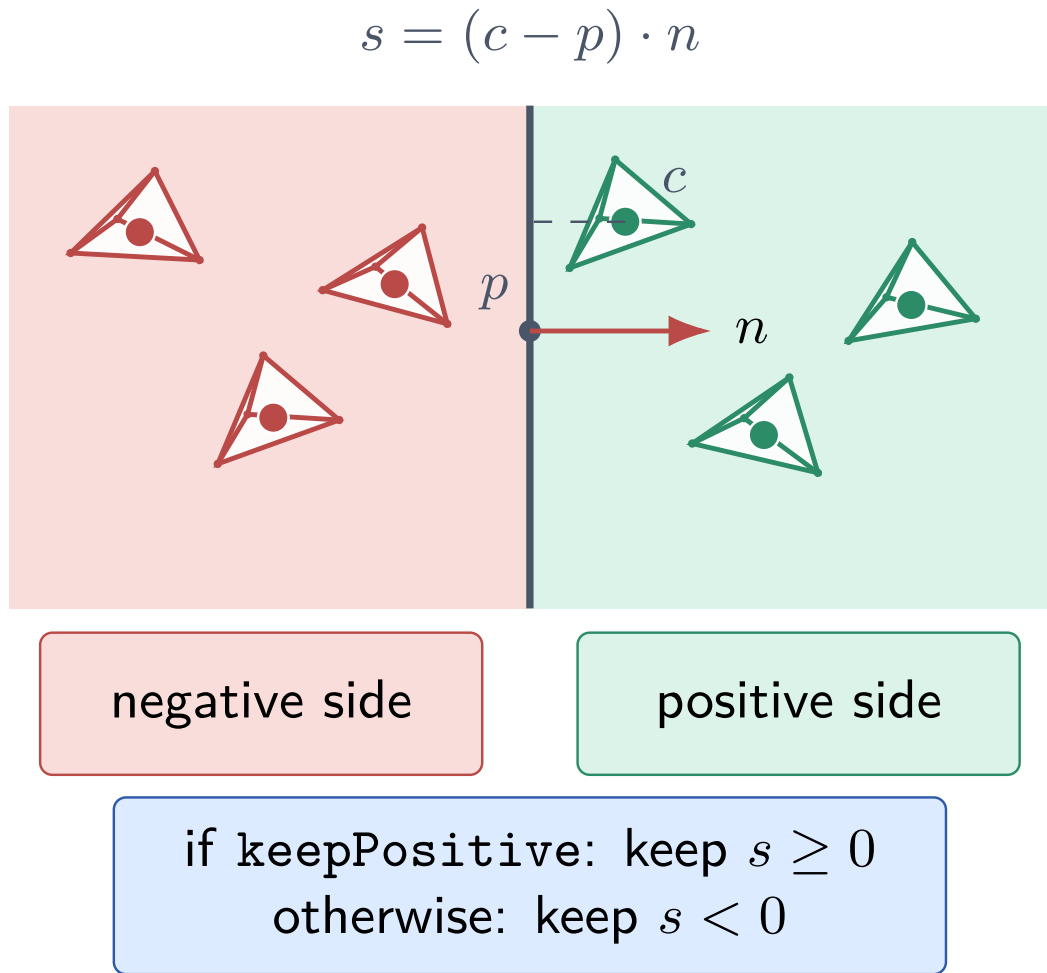


Figure 4.7: Tetrahedral cell classification by signed distance to a cutting plane. Wireframe shapes represent tetrahedral cells schematically, and colored dots mark the centroids used for classification.

This centroid-based classification is computationally simple and stable, but it is an approximation. A tetrahedron intersected by the plane is not geometrically split. Instead, its final side depends only on the position of its centroid.

4.6.2. Boundary Vertex Detection

After classifying the cells, the algorithm detects which vertices are shared between the two sides of the cut. Two boolean arrays are built: one records whether each vertex is referenced by any cell on the kept side, and the other records whether it is referenced by any cell on the cut side. A vertex marked in both arrays is a boundary vertex. Such a vertex is the reason why the two sides remain physically connected in the original mesh.

If shared vertices are left unchanged, cells on both sides still refer to the same position and velocity entries in the simulation buffers. Any force applied to one side can therefore propagate through these shared degrees of freedom. The purpose of vertex duplication is to remove this shared state while preserving the existing cell shapes as much as possible; the shared boundary vertices that must be duplicated are shown in Figure 4.8.

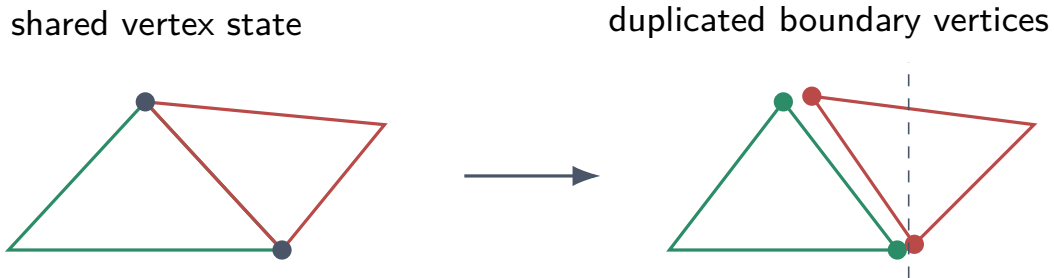


Figure 4.8: Shared vertices before and after duplication at the cut boundary.

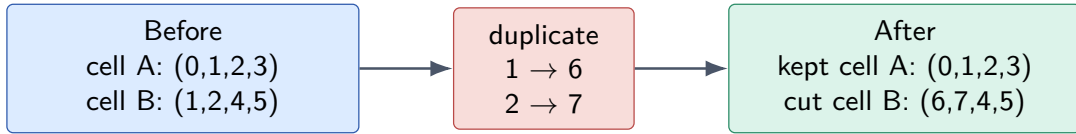
4.6.3. Vertex Duplication and Cell Remapping

For every boundary vertex, the implementation creates a duplicate vertex. The duplicate receives the same current position, rest position, and velocity as the original. It is assigned to a new vertex group representing the cut-side component. The original vertex remains associated with the kept side.

The cell index array is then remapped. For every cell classified as belonging to the cut side, each referenced boundary vertex is replaced by its duplicate. Cells on the kept side continue to use the original vertex indices. This creates two sets of tetrahedra that occupy the same geometric location along the boundary at the moment of cutting, but no longer

share the same vertex state.

This operation is the key topological step of the implementation. It does not remove tetrahedra and it does not create a new cut surface, but it disconnects the deformation graph along the boundary induced by the plane classification. After the cut, the solver can move the two sides independently because the affected cells no longer share the same vertices. The corresponding cut-side cell index remapping is illustrated in Figure 4.9.



Only cut-side cells are remapped to duplicated indices.

Figure 4.9: Cell index remapping after boundary vertex duplication.

4.6.4. GPU Buffer Rebuild

Once the new topology has been constructed on the CPU, the affected GPU buffers are rebuilt. The updated rest positions, current positions, velocities, vertex groups, rigid groups, and cell indices are uploaded to the device. Buffers whose size depends on the number of vertices are reallocated because the vertex count increases after duplication.

The simulation buffers are then reset. This includes force buffers, mass buffers, inverse mass buffers, temporary XPBD displacement buffers, selected-vertex buffers, quasi-static buffers, FEM multipliers, inverse basis matrices, and rest volumes. The active-cell buffer is initialized so that all cells are considered active after the separation update.

Reinitializing these buffers is necessary because the solver uses arrays whose dimensions and entries depend directly on the current number of vertices and cells. If any of these arrays retained the previous size or previous indexing, subsequent simulation steps could read invalid memory or apply constraints to outdated topology.

4.6.5. Embedding Recalculation

After the topology update, the Gaussian-to-tetrahedron embedding must be recomputed. The previous embedding refers to the old tetrahedral connectivity and may connect Gaussian primitives across the new seam. The implementation resets the global tetrahedron index buffer and reruns the embedding search for all tetrahedral cells. This assigns Gaus-

sians to the updated cell connectivity.

The recomputed embedding is then copied back to the CPU for seam handling. Some Gaussians may no longer have a valid embedding after the cut. These Gaussians are deactivated by setting their opacity to zero. Other Gaussians may still have an embedding, but their local/global interpolation references can span different groups after vertex duplication. These seam cases are handled by selecting a dominant group and reassigning the embedding to avoid visual bridging across the cut, which is why the Gaussian embedding must be rebuilt after the topology update as shown in Figure 4.10.

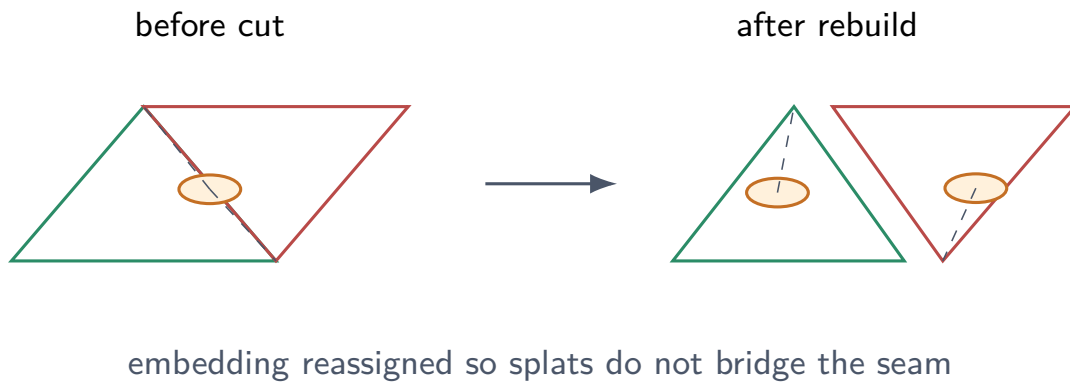


Figure 4.10: Gaussian embedding before and after the runtime topology update.

When seam Gaussians are reassigned, the implementation disables the two-level interpolation mode. This is a practical safeguard. Two-level interpolation can blend multiple embedded tetrahedral references; after a split, such blending can accidentally reconnect the visual representation across the seam. Switching to a single-corner interpretation after the cut avoids this type of bridge artifact.

4.6.6. FEM and Rigid State Reinitialization

The final stage of the cut algorithm reinitializes the physical state derived from the tetrahedral mesh. FEM bases and rest volumes are recomputed from the updated rest positions and cell indices. Vertex masses and inverse masses are also recomputed. Rigid initialization is run again so that rigid group data matches the updated vertex arrays.

This step completes the synchronization between topology, simulation, and rendering. After it finishes, the normal per-frame update loop can resume. The solver sees the updated vertex count and cell connectivity, while the renderer receives Gaussian positions and attributes derived from the updated embedding.

4.7. CUDA Helper Kernels

In addition to the main CPU-side topology update, the implementation adds CUDA helper kernels for simpler cut and deactivation operations. These kernels are implemented in `CudaKernels.cu`.

The first helper is `cut_by_plane_kernel`. It operates directly on Gaussian positions and opacities. For each Gaussian, the kernel computes the signed distance to the plane and sets the opacity to zero if the Gaussian lies on the removed side. This produces a visual cut, but it does not modify the simulation topology. For this reason, it is used only as a fallback when no tetrahedral cells are available.

The second helper is `cut_cells_by_plane_kernel`. It classifies cells by the signed distance of their centers and updates a cell activity buffer. A third helper, `deactivate_gaussians_from_cells_kernel`, deactivates Gaussians whose embedding no longer references any active cell. These kernels support the same conceptual model as the main cut operation: a plane classifies simulation elements, and Gaussian visibility is updated according to the surviving embedding. The fallback and helper kernel paths used for opacity masking and cell-based deactivation are summarized in Figure 4.11.

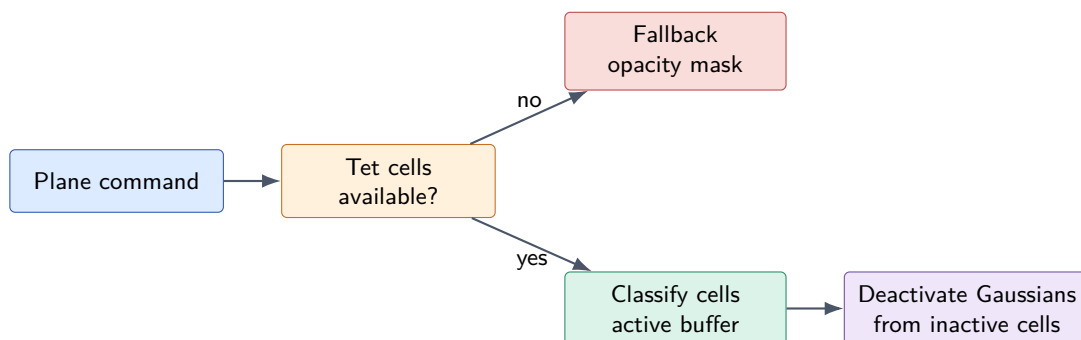


Figure 4.11: CUDA helper kernels for opacity masking and cell-based Gaussian deactivation.

4.8. Plugin Robustness and Diagnostics

Native plugin failures can be difficult to debug from Unity because exceptions, CUDA errors, and invalid parameters may cross the managed/native boundary in unhelpful ways. The implementation therefore adds additional diagnostic infrastructure to the plugin layer.

The plugin now writes log messages to a temporary log file. Important API calls, including model loading, point-of-view initialization, texture pointer retrieval, and plane cuts, are

logged. The exported functions catch standard and unknown exceptions and store a readable error message that Unity can retrieve through `GetLastMessage`. The plugin also installs an invalid-parameter handler, a terminate handler, and a vectored exception handler on Windows. These handlers are intended to capture failures that would otherwise terminate the process without a useful message. The diagnostic path from native API calls to Unity-visible error reporting is summarized in Figure 4.12.

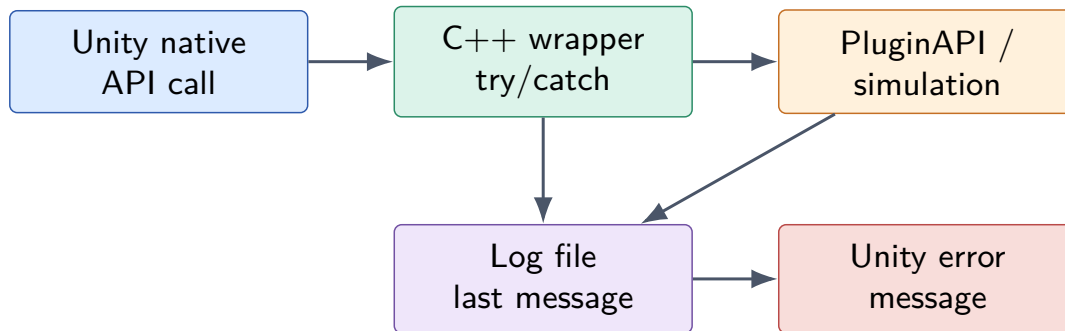


Figure 4.12: Native plugin logging and exception reporting path.

The loading path was also made more explicit. The model path is validated before loading, and the number of object offsets is checked against the number of object material entries. If model files are found on disk but Unity has not provided enough object configuration entries, loading fails with a specific message. The load method also tracks the current loading stage and includes it in the error message if an exception occurs.

4.9. Build Workflow

The native plugin is built with CMake and Visual Studio. The build script `build.bat` was updated to make the process more reproducible. It defines the build directory, generator, CUDA architecture, CUDA runtime mode, and build configuration explicitly. After building, it copies the generated `gaussiansplatting.dll` into the Unity plugin folder. If an older DLL is already present, it is copied to `gaussiansplatting.lastknown.dll` before being replaced. When a program database file is available, it is copied as well, which is useful for native debugging.

This workflow reduces the risk of accidentally running an outdated plugin inside Unity. It also provides a simple recovery path through the last-known DLL if a new native build fails at runtime.

5 | Experimental Validation

This chapter validates the runtime separation pipeline described in Chapter 4. The goal of the validation is not to prove that the implemented cut is a complete surgical remeshing algorithm. Instead, the experiments verify the specific objective of this thesis: enabling an interactive plane-based separation operation on a deformable Gaussian Splatting object while preserving a consistent connection between user interaction, physical simulation, and rendering.

The validation is organized around three questions. First, the native implementation must produce a meaningful topological separation when the plane intersects the tetrahedral physical structure. Second, the two resulting parts must behave independently under subsequent interaction, showing that the cut changed the physical simulation state and not only the visual opacity of the Gaussians. Third, the operation must remain compatible with the real-time rendering and simulation loop used by the original VR-GS system.

5.1. Validation Setup

The experiments are performed on the modified VR-GS project branch developed for this thesis. The Unity application loads a Gaussian Splatting model together with its tetrahedral physical structure and starts the native C++/CUDA plugin. The runtime separation pipeline is available from the VR interaction flow and from the desktop interaction flow: both modes submit a plane definition to the same managed wrapper and native `CutByPlane` implementation. The desktop mode is used for most screenshots and repeated checks only for convenience, because it makes camera placement, plane adjustment, and image capture easier to repeat.

The tested scene is the deformable body example used for validation. The scene contains the Gaussian representation, the associated tetrahedral physical structure, and the simulation data required for FEM/XPBD deformation. This choice is aligned with the intended medical-simulation direction of the work: the body model is used as a representative anatomical scene for testing the runtime separation pipeline, not as evidence of

clinical validity. The relevant validation actions are:

- inspecting the model from the active viewpoint;
- activating cut-plane selection;
- translating and rotating the cut plane;
- executing the cut while choosing which side of the plane remains in the original group;
- manipulating the separated parts through the available interaction controls;
- recording qualitative screenshots and profiling output.

The validation environment is reported through the following hardware and software configuration:

Table 5.1: Hardware and software configuration for the validation.

Item	Value
CPU	AMD Ryzen 5 3600, 6 cores / 12 threads, 3.6 GHz
GPU	NVIDIA GeForce RTX 3060 Ti, 8 GB VRAM
RAM	16 GB, 15.92 GB usable
Operating system	Microsoft Windows 11 Home, 64-bit, version 10.0.26200
Unity version	Unity 2022.3.9f1
CUDA version	CUDA 13.1, nvcc V13.1.115, NVIDIA driver 591.86
VR headset	Meta Quest 2
Dataset/model	body validation scene: 82,090 Gaussians, 734 tetrahedral-structure vertices, 932 surface faces, 2,891 tetrahedral cells

5.2. Functional Validation Method

The functional validation checks that each part of the pipeline is connected to the next one. A successful cut requires the following chain to complete:

1. the Unity interaction layer computes a plane point and normal in world coordinates;
2. the managed C# wrapper converts the plane into the coordinate system used by the native plugin;
3. the exported native function receives the plane and dispatches it to the active Gaussian Splatting instance;

4. the tetrahedral cells are classified with respect to the plane;
5. vertices shared by cells on opposite sides are duplicated;
6. cut-side cells are remapped to the duplicated vertices and assigned to a new group;
7. GPU buffers and simulation bases are rebuilt;
8. Gaussian embedding data are recomputed or invalid Gaussians are deactivated;
9. subsequent interactions affect the separated groups independently.

The validation therefore combines visual inspection with specific observable effects. The most important effect is not the visual disappearance of a set of Gaussians, but the change in physical connectivity. If the object is only visually masked, the virtual hands can still transmit deformation across the hidden interface. If the implemented topological update is active, pulling or pushing one side of the model after the cut no longer forces the other side to follow as part of the same connected deformable body. The complete validation sequence used to inspect this behavior is shown in Figure 5.1.

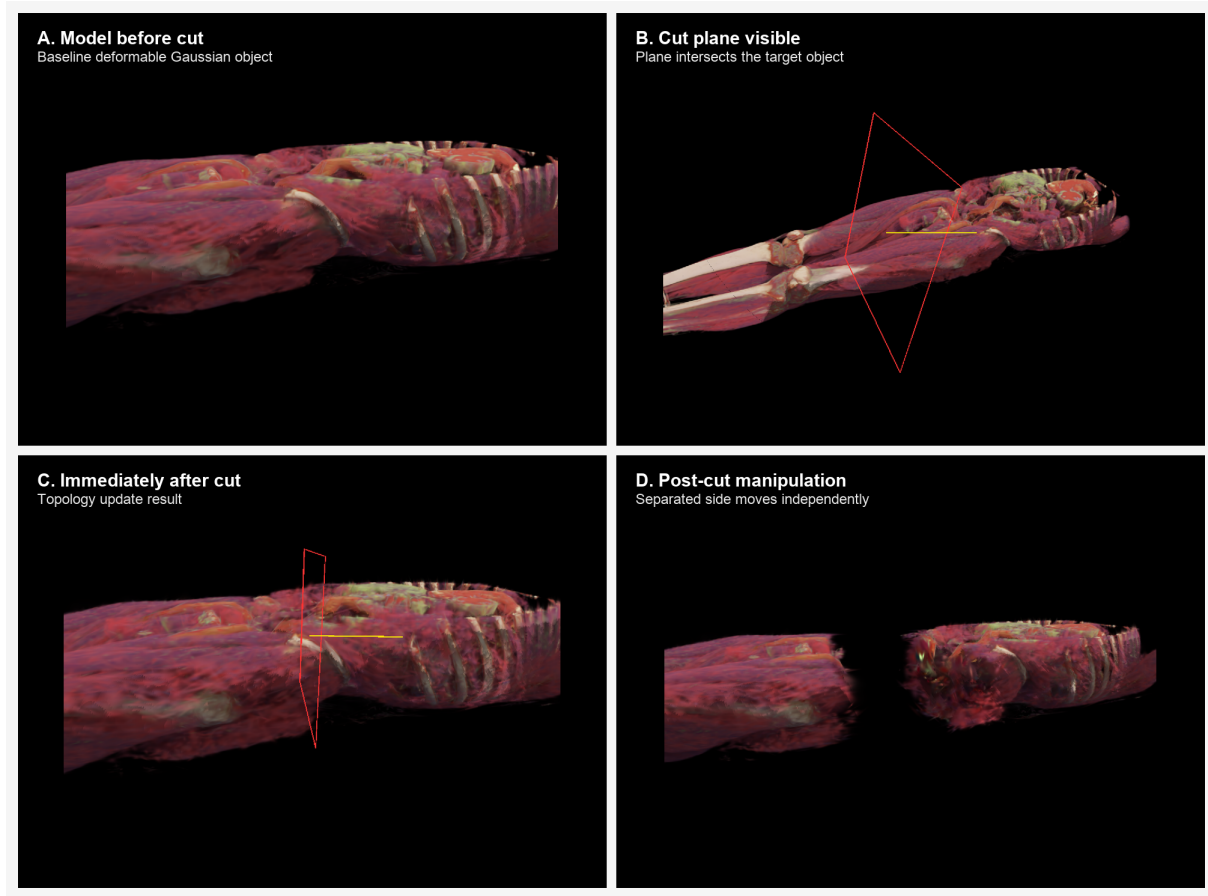


Figure 5.1: Validation workflow for the runtime cut operation: model before the cut, cut-plane preview, immediate result after separation, and post-cut manipulation.

5.3. Cut-Plane Interaction Workflow

The first validation experiment focuses on the user-facing cut selection workflow. The system must let the user define a plane, preview its position with respect to the deformable object, execute the separation command, or cancel the operation without changing the simulation state. In VR, the plane can be driven through the immersive interaction flow. In desktop mode, the same operation is exposed through keyboard and mouse controls, which makes repeated validation and screenshot capture more convenient while preserving the same native separation path; the preview state used before executing the validation cut is shown in Figure 5.2.

The interaction layer is considered valid if the following criteria are satisfied:

- the cut-plane preview appears only while cut selection is active;
- plane position and orientation can be adjusted in a stable way;

- pressing the cut command invokes the native `CutByPlane` call;
- cancelling the operation exits cut mode without modifying the model.

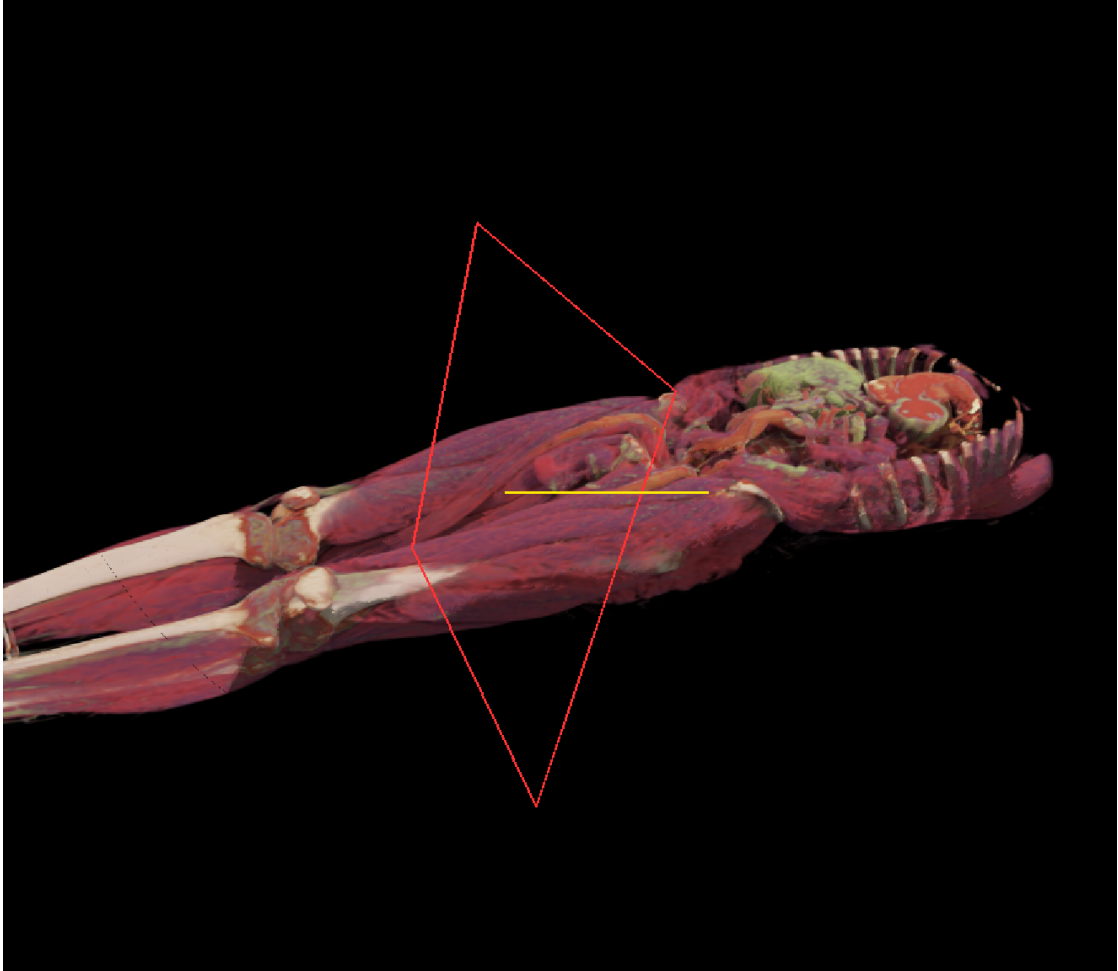


Figure 5.2: Validation screenshot of the cut-plane preview before execution. The plane intersects the deformable object before the separation command is applied.

5.4. Topological Separation Validation

The second experiment verifies that the native operation changes the tetrahedral simulation topology. For each tested plane, the implementation classifies tetrahedral cells by evaluating the signed distance of the cell center from the plane. Cells on the selected side remain in the original group, while cells on the opposite side are assigned to a new group. Vertices that are referenced by cells on both sides are duplicated, and the cut-side cells are remapped to those duplicated vertices.

This operation can be validated through three observable conditions. First, the native

call should return a non-zero value when the plane intersects the object and produces duplicated boundary vertices or deactivated seam Gaussians. Second, the visual result should show a separation along the selected region instead of only a global opacity mask. Third, after the operation, subsequent deformation should no longer propagate across the separated boundary as it did before the cut. The before-and-after comparison in Figure 5.3 documents this visual transition from the same camera viewpoint.

The following table summarizes the planned functional test matrix.

Table 5.2: Functional validation matrix for the runtime cut operation.

Test	Procedure	Expected result
No-op plane	Place the plane outside the object and execute the cut.	The native call returns zero and the object remains unchanged.
Intersecting plane	Place the plane through the object and execute the cut.	The native call returns a positive value and the seam changes visually.
Side selection	Repeat the same cut with and without the positive-side modifier.	The group assignment is inverted consistently with the selected side.
Post-cut manipulation	Pull one side of the separated model with a virtual hand.	The selected side moves independently from the other side.
Repeated cut	Execute a second plane cut after the first separation.	The system remains stable or reports a controlled no-op for invalid cuts.
Degenerate plane	Send a near-zero normal vector.	The native function rejects the operation and returns zero.

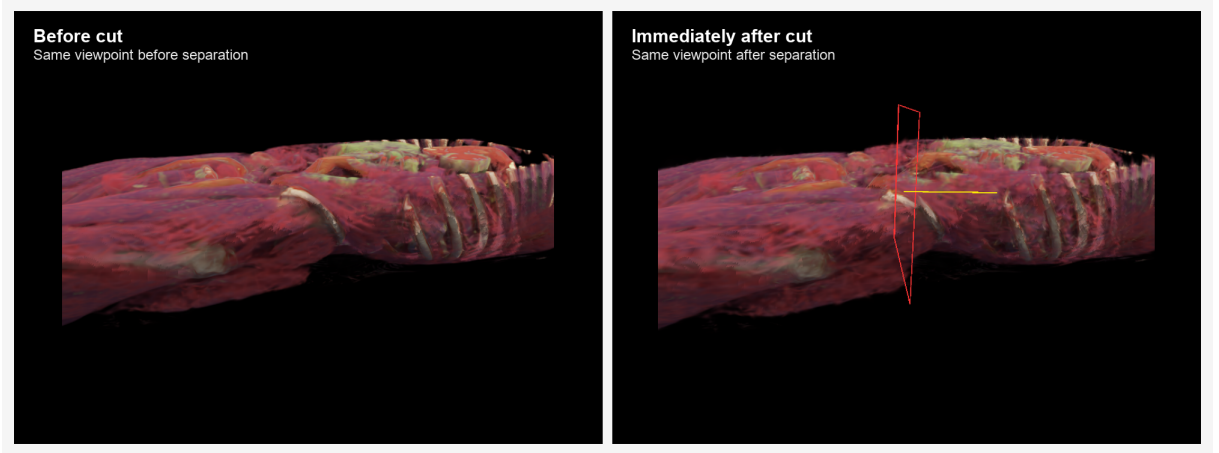


Figure 5.3: Before-and-after result of a plane-based separation from the same camera viewpoint.

5.5. Rendering and Embedding Consistency

The cut operation affects both simulation and rendering data. The tetrahedral physical structure is updated by vertex duplication and cell remapping, while Gaussians still need valid embedding data to follow the new deformed state. For this reason, the validation must check that the rendered splats remain attached to the correct side of the separated physical structure.

The implementation rebuilds the global tetrahedral embedding after the topology change. Gaussians that can no longer be assigned to a valid tetrahedral configuration are deactivated by setting their opacity to zero. In addition, when seam Gaussians are reassigned, the implementation disables the two-level local-global interpolation path and switches to a safer single-corner interpolation behavior. This avoids visible bridging artifacts where Gaussians near the cut could continue blending across both sides of the separated physical structure.

The consistency validation, illustrated after post-cut manipulation in Figure 5.4, inspects the following visual cases:

- no large disconnected cluster of Gaussians remains attached to the wrong side of the cut;
- no persistent visual bridge remains across the separated boundary;
- manipulating one side updates the Gaussians embedded in that side;
- seam Gaussians do not produce unstable flickering after the rebuild;

- the rendered model remains visible after the first simulation step following the cut.



Figure 5.4: Gaussian embedding behavior after the topology update. The separated regions remain visually disconnected during post-cut manipulation.

5.6. Performance Measurement

The project already contains profiling hooks in the native plugin. When profiling is enabled, selected CUDA operations are wrapped by a macro that synchronizes the device before and after the measured call. The elapsed time is recorded with a microsecond clock and accumulated into counters for collision detection, FEM solving, collision solving, XPBD updates, embedding, shadow rendering, and left/right-eye rendering. After 1000 simulation steps, the code writes average counter values to a `sim_time.txt` file.

These counters identify the main runtime costs of the simulation and rendering loop. The performance evaluation compares two conditions. The first condition measures the uncut model during normal interaction. The second condition measures the same scene after one successful runtime separation. Comparing these conditions indicates whether the additional vertices, rebuilt embedding data, and disabled two-level interpolation path change the runtime profile in a measurable way.

Table 5.3: Performance counters for the validation scene.

Metric	Before cut	After cut	Difference
Average simulation step time	6.888 ms	8.511 ms	+1.623 ms
Collision detection	3.533 ms	3.922 ms	+0.389 ms
FEM solve	1.032 ms	1.132 ms	+0.100 ms
XPBD update	2.122 ms	2.378 ms	+0.256 ms
Gaussian embedding update	0.117 ms	0.118 ms	+0.001 ms
Left-eye rendering	1.005 ms	1.218 ms	+0.213 ms
Right-eye rendering	1.006 ms	1.218 ms	+0.212 ms
Average frame rate	102.6 FPS	84.7 FPS	-17.9 FPS

5.7. Robustness Tests

Robustness is important because the cut operation changes memory ownership and GPU buffer sizes at runtime. The implementation includes several guards that are validated in this section. Null plane inputs, zero-length normals, empty models, planes that do not split the tetrahedral cells, and cuts that do not remap any shared vertex references are treated as no-op cases. In these situations, the native function returns zero instead of attempting to rebuild the simulation state.

Runtime exception handling is also part of the validation. The exported plugin entry point catches standard and non-standard exceptions, stores a diagnostic message, and writes plugin logs on Windows. This is useful during Unity testing because a native exception would otherwise be difficult to diagnose from the managed side. The validation confirms that controlled failures produce a Unity log message and do not leave the application in a corrupted state.

The following robustness checks are documented with short observations:

- no crash when cutting outside the model;
- no crash when cancelling cut mode repeatedly;
- no crash when manipulating after a successful cut;
- no crash when saving the model after a cut;
- clear diagnostic log if the native plugin rejects a cut;
- stable behavior when the selected plane intersects only a small part of the tetrahedral physical structure.

5.8. Qualitative Results

The qualitative result expected from the implemented system is a visible and interactive separation of the deformable object. Before the cut, the body reacts as a single connected simulated object. After the cut, one side can be selected and moved while the other side remains independent. This difference is the main evidence that the implementation changes the physical representation, not only the rendered opacity.

The qualitative result is shown in Figure 5.5 and summarized by a four-step sequence:

1. the original deformable Gaussian Splatting model;
2. the same model with the selected plane overlay;
3. the immediate result after the runtime separation;
4. post-cut manipulation showing independent motion of the separated side.

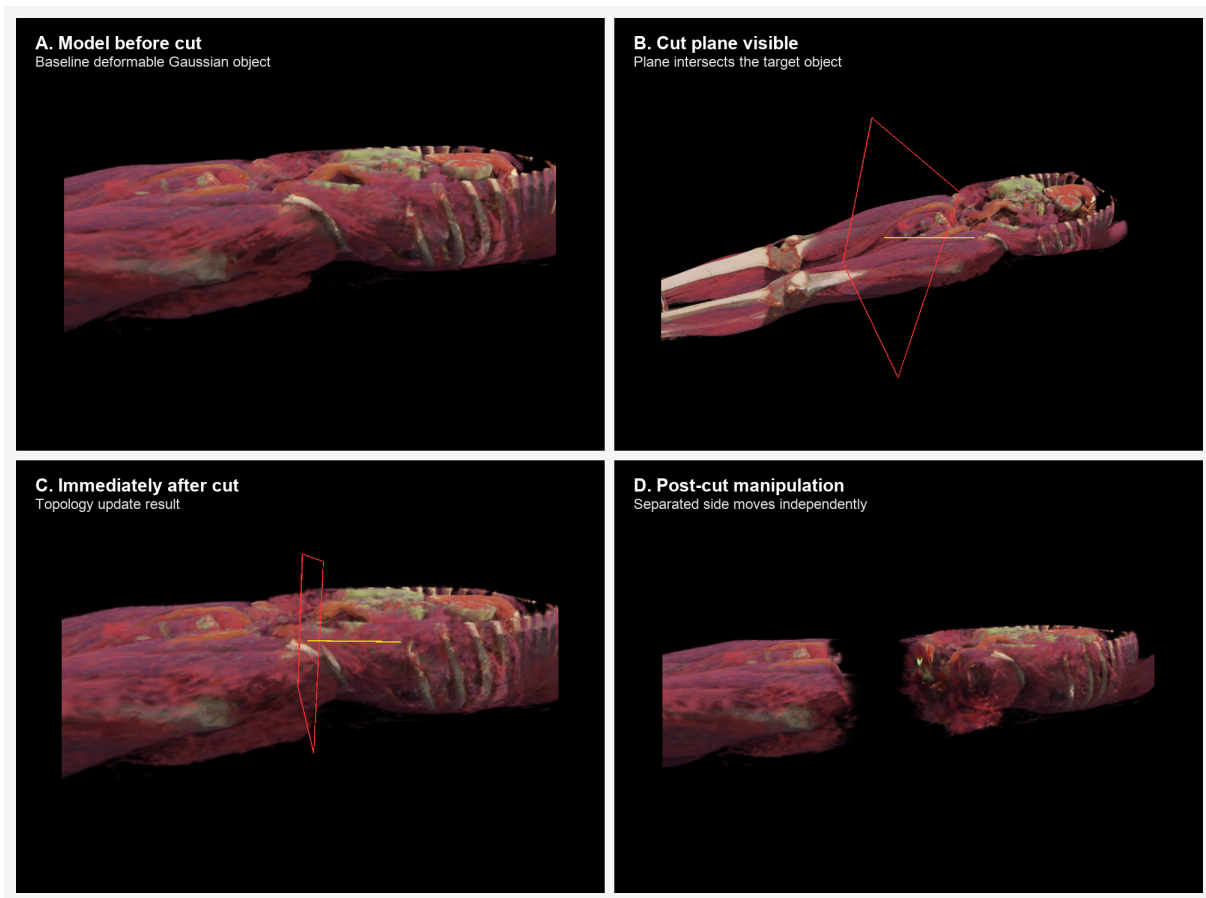


Figure 5.5: Qualitative result sequence for the implemented runtime separation: original model, selected plane, result after cut, and independent manipulation after the cut.

5.9. Results

The validation confirms the main experimental result of the thesis: a Gaussian Splatting object embedded in a tetrahedral physical structure can be separated at runtime by a user-defined plane, and the resulting parts can be manipulated as physically independent regions. In the validation scene, this behavior is demonstrated on a body model chosen as a representative anatomical object for the intended medical-simulation application area. The same native separation pipeline is used by the VR and desktop interaction flows; the desktop flow is used in this chapter only to make repeated captures and measurements easier to perform. The observed behavior depends on an actual topology update, because post-cut interaction no longer propagates deformation across the separated boundary as it does before the cut.

The performance measurements show that the operation remains compatible with the real-time simulation and rendering loop of the validation scene. The average simulation step increases from 6.888 ms before the cut to 8.511 ms after the cut, while the average frame rate changes from 102.6 FPS to 84.7 FPS. The largest measured cost remains collision detection, followed by XPBD and FEM updates. The Gaussian embedding update remains almost unchanged in the measured profile, indicating that the additional runtime cost is mainly associated with the updated simulation state and rendering workload after separation.

The results also show the practical boundaries of the current implementation. The separation boundary follows the resolution of the tetrahedral physical structure because cells are classified by their centers; seam appearance depends on the available Gaussian distribution and on embedding repair; and the runtime split involves GPU buffer reallocation and simulation reinitialization. These points are discussed together with the broader future-work directions in Section 6.1.

6 | Conclusions and Future Work

This thesis presented an extension of a deformable Gaussian Splatting viewer with runtime plane-based separation. Starting from a physically aware VR-GS pipeline, the work added the ability to define a cut plane from Unity, forward it to the native C++/CUDA plugin, modify the tetrahedral physical structure, and rebuild the data needed to keep Gaussian rendering consistent with the updated simulation state.

The work was motivated by a broader application direction in interactive medical simulation. In that domain, visually realistic anatomical models are useful only if they can also support physically meaningful interaction. Runtime deformation and separation are therefore relevant building blocks for future training or simulation tools, even though the system developed in this thesis was evaluated as a technical prototype and not as a clinically validated medical simulator.

The main result is an interactive operation that goes beyond visual masking. A pure opacity-based cut can hide part of a Gaussian model, but it does not change the mechanical connection inside the simulation. The implemented method instead operates on the tetrahedral physical structure: cells are classified with respect to the plane, shared boundary vertices are duplicated, cut-side cells are remapped to the duplicates, and the separated side is assigned to a new simulation group. After the update, subsequent interaction can move the separated regions independently, which is the central behavioral goal of the thesis.

The implementation also shows that adding topology changes to a Gaussian Splatting system requires coordination across multiple layers. The Unity frontend must provide an interaction model for selecting the edit operation. The managed wrapper must expose the native function safely and convert coordinates correctly. The plugin must update CPU and GPU data structures without leaving the solver in an inconsistent state. Finally, the renderer must receive valid embedding data so that the visual Gaussians remain attached to the updated physical structure. Treating the cut as a whole-system operation is therefore more important than modifying any single buffer in isolation.

The first contribution is the interaction workflow for plane-based separation. The original

system targets VR interaction, and the implemented cut path remains available through the same simulation backend. A desktop mode was also added to support development, debugging, repeated validation, and screenshot collection without requiring headset setup during every test.

The second contribution is the Unity-to-native cut API. A new `CutByPlane` call is exposed by the native plugin and called from the C# viewer code. The wrapper converts the point and normal from Unity world space to the coordinate convention used by the plugin, including the case where the model uses a different up-axis convention.

The third contribution is the runtime separation algorithm. The method performs cell classification, shared-vertex detection, vertex duplication, cell remapping, group reassignment, GPU buffer rebuilding, and simulation reinitialization. This allows the tetrahedral physical structure to represent two physically independent regions after the edit.

The fourth contribution is the repair of the rendering-simulation coupling after the topology update. The implementation recomputes global tetrahedral embedding information, deactivates invalid seam Gaussians, and disables the interpolation path that can visually bridge the newly separated boundary. These steps preserve the consistency between the physical structure and the rendered Gaussian model.

6.1. Limitations and Future Work

The implemented method focuses on runtime plane-based separation of an already reconstructed and simulated Gaussian Splatting object. Gaussian reconstruction, object segmentation, tetrahedral mesh generation, and material estimation are treated as inputs provided by the existing VR-GS pipeline. Extending the system to cover these stages would require additional reconstruction, preprocessing, and material-identification components.

The validation uses a body model because anatomical scenes are the intended application direction for this line of work. However, the experiments do not evaluate surgical realism, anatomical correctness, user training outcomes, or clinical safety. Those aspects would require domain-specific datasets, task-specific protocols, expert review, and validation metrics beyond the scope of this thesis.

The separation itself is intentionally approximate. It classifies whole tetrahedral cells by their centroid position relative to the cut plane. It does not compute exact plane-tetrahedron intersections, does not generate new tetrahedra along the cut, and does not create an explicit cut-surface mesh. As a result, the separation boundary is constrained

by the resolution and arrangement of the existing tetrahedral physical structure. A finer structure would provide a more accurate boundary, but it would also increase simulation cost.

Seam quality is another limitation. Gaussian Splatting is not an explicit surface-mesh representation, so newly exposed interior surfaces are not automatically available after separation. The implementation recomputes embedding data and deactivates invalid Gaussians, but local quality near the boundary depends on the density of the Gaussians, the resolution of the tetrahedral physical structure, and the embedding configuration. This can lead to visual thinning or small discontinuities near the separated region.

The current implementation also keeps the separated regions inside the same global simulation object. The regions can be disconnected at the vertex level, but the system does not compact connected components into independent object instances. It also does not provide persistent editing histories, undo and redo, or exact collision handling between newly exposed surfaces. These features would be required for a more complete interactive editing system.

The runtime cost of the edit operation is another area for improvement. The current split performs CPU-side topology processing, data transfers between CPU and GPU, GPU buffer reallocation, and simulation reinitialization. This is acceptable for occasional editing operations, but it is not optimized for continuous high-frequency cutting or very large models.

The most direct future extension is exact cut-surface generation. Instead of classifying tetrahedra as whole cells, the system could compute the intersection between the cut plane and each tetrahedron. New vertices and tetrahedral elements could then be created around the intersection, producing a boundary that follows the user-defined plane more accurately.

A second direction is better seam rendering. Future work could synthesize additional Gaussians along the cut boundary, redistribute existing Gaussians near the seam, or combine the Gaussian representation with a lightweight explicit surface for newly exposed regions.

A third direction is a GPU-centered topology update. Moving classification, duplication, remapping, and embedding repair to GPU kernels would reduce transfer overhead and make the approach more suitable for frequent editing.

The interaction model could also be extended. The current operation uses a single plane, which is simple and predictable. More expressive tools could allow curved cuts, stroke-

based cuts, volumetric selection, brush-based deletion, or multi-step editing workflows. In VR, the plane could be attached directly to a controller or represented as a virtual blade. Finally, the system could be evaluated with a broader set of anatomical models and user tasks. The current validation focuses on the body example used during development. Testing different Gaussian densities, tetrahedral physical structure resolutions, material parameters, and interaction scenarios would clarify when the approximate separation method is sufficient and when a more exact remeshing approach is necessary. For medical-simulation applications, future work should also include expert-defined tasks and evaluation criteria that reflect the requirements of training or procedural simulation.

6.2. Final Remarks

The thesis demonstrates that runtime physical editing of Gaussian Splatting objects is possible when the visual representation is coupled to a deformable simulation structure. The implemented cut operation is not a complete solution to general topological editing, but it establishes a concrete path for combining photorealistic Gaussian rendering with interactive physical separation. This is a useful step toward richer editing workflows for reconstructed anatomical content in virtual and mixed reality environments, and toward future medical simulation systems that require both visual realism and physically meaningful interaction.

Bibliography

- [1] J. Bender, M. Müller, and M. Macklin. A survey on position based dynamics. In *EG 2017 – Tutorials*. The Eurographics Association, 2017. doi: 10.2312/egt.20171034. URL <https://doi.org/10.2312/egt.20171034>.
- [2] D. Bielser, V. A. Maiwald, and M. H. Gross. Interactive cuts through 3-dimensional soft tissue. *Computer Graphics Forum*, 18(3):31–38, 1999. doi: 10.1111/1467-8659.00325. URL <https://doi.org/10.1111/1467-8659.00325>.
- [3] D. Bielser, P. Glardon, M. Teschner, and M. H. Gross. A state machine for real-time cutting of tetrahedral meshes. *Graphical Models*, 66(6):398–417, 2004. doi: 10.1016/j.gmod.2004.05.009. URL <https://doi.org/10.1016/j.gmod.2004.05.009>.
- [4] S. Cotin, H. Delingette, and N. Ayache. A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation. *The Visual Computer*, 16(8):437–452, 2000. doi: 10.1007/PL00007215. URL <https://doi.org/10.1007/PL00007215>.
- [5] 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. doi: 10.48550/arXiv.2401.16663. URL <https://arxiv.org/abs/2401.16663>.
- [6] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023. doi: 10.1145/3592433. URL <https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/>.
- [7] M. Macklin, M. Müller, and N. Chentanez. XPBD: Position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*, MIG ’16, pages 49–54. ACM, 2016. doi: 10.1145/2994258.2994272. URL <https://doi.org/10.1145/2994258.2994272>.
- [8] U. Meier, O. López, C. Monserrat, M.-C. Juan, and M. Alcañiz Raya. Real-time deformable models for surgery simulation: A survey. *Computer Methods and Pro-*

- grams in Biomedicine*, 77(3):183–197, 2005. doi: 10.1016/j.cmpb.2004.11.002. URL <https://doi.org/10.1016/j.cmpb.2004.11.002>.
- [9] 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 *Computer Vision – ECCV 2020*, pages 405–421. Springer, 2020. doi: 10.1007/978-3-030-58452-8_24. URL <https://arxiv.org/abs/2003.08934>.
- [10] N. Molino, Z. Bao, and R. Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Transactions on Graphics*, 23(3):385–392, 2004. doi: 10.1145/1015706.1015734. URL <https://doi.org/10.1145/1015706.1015734>.
- [11] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Proceedings of the Third Workshop on Virtual Reality Interactions and Physical Simulations*, VRIPHYS 2006, pages 71–80. Eurographics Association, 2006. doi: 10.2312/PE/vriphys/vriphys06/071-080. URL <https://doi.org/10.2312/PE/vriphys/vriphys06/071-080>.
- [12] E. Sifakis, K. G. Der, and R. Fedkiw. Arbitrary cutting of deformable tetrahedralized objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 73–80. Eurographics Association, 2007. URL <https://pages.cs.wisc.edu/~sifakis/papers/cutting.pdf>.
- [13] R. W. Sumner, J. Schmid, and M. Pauly. Embedded deformation for shape manipulation. *ACM Transactions on Graphics*, 26(3), 2007. doi: 10.1145/1276377.1276478. URL <https://doi.org/10.1145/1276377.1276478>.
- [14] G. Wu, T. Yi, J. Fang, L. Xie, X. Zhang, W. Wei, W. Liu, Q. Tian, and X. Wang. 4d gaussian splatting for real-time dynamic scene rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 20310–20320, 2024. URL https://openaccess.thecvf.com/content/CVPR2024/html/Wu_4D-Gaussian_Splatting_for_Real-Time-Dynamic-Scene-Rendering_CVPR_2024_paper.html.
- [15] Z. Yang, X. Gao, W. Zhou, S. Jiao, Y. Zhang, and X. Jin. Deformable 3d gaussians for high-fidelity monocular dynamic scene reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 20331–20341, 2024. URL https://openaccess.thecvf.com/content/CVPR2024/html/Yang_Deformable_3D-Gaussians_for_High-Fidelity_Monocular-Dynamic-Scene-Reconstruction_CVPR_2024_paper.html.
- [16] J. Zhang, Y. Zhong, and C. Gu. Deformable models for surgical simulation: A survey.

IEEE Reviews in Biomedical Engineering, 11:143–164, 2018. doi: 10.1109/RBME.2017.2773521. URL <https://doi.org/10.1109/RBME.2017.2773521>.

List of Figures

2.1	3D Gaussian Splatting rendering pipeline.	6
2.2	VR-GS style architecture with Gaussian rendering and a tetrahedral physical structure.	8
2.3	Taxonomy of deformable simulation methods.	9
2.4	High-resolution visual representation embedded in a coarse tetrahedral physical structure.	10
2.5	Comparison between visual masking, element removal, vertex duplication, and exact remeshing.	11
3.1	Parameters of a 3D Gaussian primitive.	16
3.2	Projection of a 3D Gaussian ellipsoid to a 2D splat footprint.	17
3.3	Tetrahedral cell with four vertices in rest and deformed configurations. . .	18
3.4	Gaussian primitive embedded into a tetrahedral physical structure.	19
3.5	XPBD/FEM update loop used by the tetrahedral physical simulation. . . .	21
3.6	Plane classification of tetrahedral cell centroids. Wireframe shapes represent tetrahedral cells schematically, and colored dots mark the centroids used for classification.	23
3.7	Topological separation through boundary vertex duplication.	24
3.8	Data dependencies that must remain consistent after a cut.	25
4.1	Global architecture of the Unity frontend, native plugin, CUDA simulation, and Gaussian renderer.	28
4.2	Loading pipeline for segmented Gaussian objects and tetrahedral physical structures.	29
4.3	Base per-frame update pipeline before the cutting extension.	30
4.4	Desktop-mode runtime view of the loaded deformable Gaussian object. . .	31
4.5	Cut-plane preview shown before applying the separation operation. The red outline shows the selected plane intersecting the deformable object. . .	32
4.6	API flow from Unity cut-plane interaction to native simulation update. . .	33

4.7	Tetrahedral cell classification by signed distance to a cutting plane. Wire-frame shapes represent tetrahedral cells schematically, and colored dots mark the centroids used for classification.	35
4.8	Shared vertices before and after duplication at the cut boundary.	36
4.9	Cell index remapping after boundary vertex duplication.	37
4.10	Gaussian embedding before and after the runtime topology update.	38
4.11	CUDA helper kernels for opacity masking and cell-based Gaussian deactivation.	39
4.12	Native plugin logging and exception reporting path.	40
5.1	Validation workflow for the runtime cut operation: model before the cut, cut-plane preview, immediate result after separation, and post-cut manipulation.	44
5.2	Validation screenshot of the cut-plane preview before execution. The plane intersects the deformable object before the separation command is applied.	45
5.3	Before-and-after result of a plane-based separation from the same camera viewpoint.	47
5.4	Gaussian embedding behavior after the topology update. The separated regions remain visually disconnected during post-cut manipulation.	48
5.5	Qualitative result sequence for the implemented runtime separation: original model, selected plane, result after cut, and independent manipulation after the cut.	50

List of Tables

2.1	Comparison of related representation, deformation, and cutting approaches.	13
5.1	Hardware and software configuration for the validation.	42
5.2	Functional validation matrix for the runtime cut operation.	46
5.3	Performance counters for the validation scene.	49