Bachelor's thesis

Information and Communications Technology

2025

Oskar Kjeldsen

# Developing a Retopology-Based Destruction System for Unity

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2025 | 36 pages

Oskar Kjeldsen

# Developing a Retopology-Based Destruction System for Unity

This thesis focuses on creating a lightweight procedural destruction system for the Unity game engine, to enable real-time fragmentation of three-dimensional objects during runtime. The idea for the project arose from limitations of traditional destruction methods which rely on pre-factured models or resource heavy physics simulations, increasing the development time and reducing flexibility.

The project used a simplified Voronoi algorithm to group mesh triangles based on their proximity to randomly scattered seed points. A center-based extension technique was used to keep each fragment closed and physically stable. Lightweight procedural retopology principles were used to ensure that the generated meshes are optimized.

The developed system was tested by using primitive Unity models while the performance was evaluated by monitoring metrics such as frame rates, triangle counts and physical behavior. The results showed that the system successfully generated fragments that were both visually convincing and structurally complete while being lightweight for the computational power. The developed solution demonstrates itself as a balanced option between scalability, visual realism and runtime efficiency.

Keywords:

Unity, Destruction, Voronoi, Retopology

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2025 | 36 sivua

Oskar Kjeldsen

# Retopologiaan perustuvan hajoamisjärjestelmän kehittäminen Unity-pelimoottoriin

Hajoamisalgoritmit perustuvat usein valmiiksi hajotettuihin malleihin ja raskaisiin fysiikkasimulaatioihin, jotka vaativat paljon resursseja. Näissä menetelmissä ongelmaksi nouseekin kehitysprosessin joustavuuden heikentyminen ja kehitysajan pidentyminen. Tästä syystä opinnäytetyössä Unity-pelimoottorille kehitettiin kevyt hajoamisjärjestelmä, jonka avulla on mahdollista rikkoa kolmiulotteisia esineitä pelin aikana.

Opinnäytetyössä käytettiin yksinkertaistettua Voronoi-algoritmia, joka ryhmitteli esineen pintaverkon kolmiot satunnaisiin pisteisiin niiden etäisyyden perusteella. Kevyen repotologian ja kolmioryhmien keskipisteeseen venyttämisen avulla saatiin luotua vakaita ja ehjiä palasia esineestä.

Järjestelmän toimivuutta testattiin käyttämällä Unitystä löytyviä yksinkertaisia primitiivimalleja. Suorituskykyä arvioitiin seuraamalla mittareita, kuten ruudunpäivitysnopeutta, kolmioiden määrää sekä fysiikan käyttäytymistä. Tulokset osoittivat, että järjestelmä pystyi tuottamaan kappaleita, jotka ovat sekä visuaalisesti vakuuttavia että rakenteellisesti eheitä, jotka säilyvät laskennallisesti kevyinä. Kehitetty ratkaisu toimii siis tasapainoisena välimuotona skaalautuvuuden, visuaalisen realismin ja reaaliaikaisen tehokkuuden välillä.

Asiasanat:

unity, hajoamisjärjestelmä, voronoi, retopologia

# Contents

# Figures



# Tables

# List of abbreviations and terms

| | |
|---|---|
| 2D | Two-dimensional |
| 3D | Three-dimensional |
| AR | Augmented Reality |
| FPS | Frames Per Second |
| HDRP | High Definition Render Pipeline |
| IDE | Integrated Development Environment |
| QEM | Quadric Error Metrics |
| URP | Universal Render Pipeline |
| UV | Texture Coordinate Mapping |
| VR | Virtual Reality |

# 1 Introduction

In recent years, destruction effects inside video games and other interactive applications have become an increasingly important visual feature drawing users deeper into the story. However, such features are usually achieved by pre-fracturing the 3D model in question (Ronnegren, 2020). The reason for using this method is because real-time fracturing is usually more taxing for the CPU. This method, however, makes many of the destructions of objects look identical to each other. It also increases the workload for developers, especially at the independent development level.

The object of this thesis is the development of a Unity system that enables procedural object destruction using a retopology inspired, Voronoi-based mesh fragmentation system. The system is designed to allow developers and users to simulate mesh breaking dynamically in the runtime with optimized performance straight in Unity.

The idea for this system comes from the need to make object destruction inside Unity more attainable and flexible. In video game development, destruction effects are either created using large, heavy physics simulations, or by pre-creating multiple versions of an object to depict various damage stages that the object can achieve in the destruction. While both methods work effectively often, they can be resource intensive and time-consuming options (Hedén & Wessman, 2023). The created system, however, seeks to offer an alternative by generating destructible effects in real-time during gameplay while still being lightweight and highly customizable.

The project system uses a simplified form of a Voronoi diagram to break 3D object into pieces. Voronoi diagrams are methods for dividing a space into regions based on proximity to a set of points known as seed points (Aurenhammer, 1991). In this case the concept is adapted to the mesh of a 3D object. The algorithm takes the triangles of the object's mesh, finds their center points, and assigns each one to the nearest seed point. As a result, clusters of

triangles form fragments of the original mesh, thus creating a convincing destruction effect.

Although this approach does not find a way to produce a perfect 3D Voronoi fracture, it does create a similar effect that is visually convincing and efficient to calculate. By relying on 2D projections and straightforward distance comparisons, it can be used in real-time without much impact on performance.

In addition to fragmentation, the created system also addresses mesh optimization, or a simplified version of retopology. When objects are broken down and apart, the resulting fragments with many extra triangles and vertices that are not needed. These additional elements can hinder performance if not handled properly. To combat this the system includes a lightweight process that identifies and removes or merges unnecessary elements to simplify the resulting fragments.

The thesis begins with an overview of foundational topics including Unity's mesh system, Voronoi diagrams, mesh simplification, and existing destruction systems. It then details the project's planning, implementation and testing phases. The final product of a Unity system is then reviewed based on performance, usability and visual results.

# 2 Unity As A Game Engine

Developing a real-time destruction system requires a game engine that supports both mesh control and high-performance runtime processing. Unity, a widely used real-time development platform and a game engine was selected for this purpose due to its mature ecosystem, efficient rendering pipelines, and robust C# scripting API. The following chapter explores Unity's relevancy in its capabilities and broader role in interactive media production. While also discussing the specific reasons behind its use for the created system.

## 2.1 Overview of Unity

Unity is a versatile widely used real-time development platform for creating interactive 2D, 3D, AR and VR experiences and applications. Since its debut in 2005 by Unity technologies, the engine has grown into one of the most popular tools for game and application development (Incredibuild, 2025). Unity offers many tools that cater to developers of all skill levels.

The graphics system is highly adaptable to different situations, allowing developers to create visually stunning environments without undermining their performance or fidelity. Its flexible rendering and lighting pipelines —specifically the Universal Render Pipeline (URP) and the High Definition Render Pipeline (HDRP) —allow developers to customize visuals according to the requirements of different platforms and project scopes (Rey, 2025).

The collision and physics systems that are built-in to Unity support both 2D and 3D applications, allowing interactions to be realistic with dynamic forces and simulations. (Unity Technologies, 2025, c)

Unity also has tools for easier animation workflow, including Mechanim for character rigging, Timeline for sequenced events, and Cinemamachine for dynamic camera control. These tools make it possible for developers to create

complex character movements and in-engine cinematics without using external tools directly in the editor.

Finally, Unity's C# scripting API provides developers with full control over objects behavior, mesh manipulation and physics simulations. This allows developers to implement advanced procedural systems entirely in code form while also having low-level access to nearly every aspect of the engine. (Unity Technologies, 2025, a)

## 2.2 Unity's Role in the Industry

Unity has established itself as a dominant tool across multiple industries and not just as a game engine. It is widely used for mobile games, console and PC titles, visualizations in architecture, automotive simulations, film production and educational tools (Rey, 2025).

One of its standout features is its cross-platform deployment capabilities which allow developers to build an application once and then deploy the build across multiple different platforms, including Android, iOS, Windows, PlayStation, Xbox and WebGL. (Unity Technologies, 2024)

Unity's modular, asset-based workflow which is supported by its extensive Asset Store and the active community lowers the barrier to entry across the development sphere for both independent and larger AAA studios (Rey, 2025).

## 2.3 Reason for Choosing Unity for this Project

Unity 6 was chosen as the development platform for this system due to its strengths in real-time mesh manipulation, runtime scripting and built-in physics.

Unity's Mesh class provides straightforward access to mesh data structures, allowing vertex level operations which are essential for procedural fragmentation and lightweight retopology. Additionally, Unity's C# environment

also supports fragmentation of the meshes of objects without the use of external libraries at runtime.

Unity’s physics engine includes ready-to-use components such as rigidbodies and colliders. This simplifies the simulation of the destroyed fragments as now the generation of the custom physics engine can be less intensive.

Unity also has extensive documentation of tutorials and information freely available to any user. This helps even a novice developer to be successful in their development process.

# 3 Theoretical Background

Simulating realistic destruction visual effects in games require manipulation of the mesh data, dividing space intelligently, and optimizing geometry in real-time. The following sections outline the mathematical theory and key concepts—mesh structures, Voronoi-based fragmentation, and procedural retopology—that form the foundation of the system used in this project.

## 3.1 Mesh Structures in Unity

In Unity, 3D objects are represented using meshes, which essentially are composed of vertices, edges, and triangles. A vertex defines a point in 3D space, and three of them connected with edges form a triangle, which is the smallest possible surface used in the creation of 3D shape, shape which is often used to define the shape of a model. In the Unity game engine this data is stored into the Mesh class, which essentially contains arrays of vertices, triangle indices, normals, UVs, and other associated data.

Mesh data can be modified in real-time using C# scripts. This enables developers to deform, procedurally generate or break objects during gameplay using functions such as Mesh.vertices, Mesh.triangles, and Mesh.RecalculateNormals(). (Unity Technologies, 2025, b)

## 3.2 Voronoi Diagrams & Their Application in Fragmentation

A Voronoi diagram is a mathematical method of dividing space into regions based on the distance to a specific set of points, which are called seed points. Each region in a Voronoi diagram contains all the points which distance to one particular seed point is shortest compared to the distances to the other seed points. Basically, if a point is closer to seed point A than to B or C, it belongs to seed point A’s region. Originally the concept comes from computational

geometry and has many applications, including spatial analysis, procedural generation and mesh segmentation (Aurenhammer, 1991).

In the created system, the Voronoi diagrams are used to calculate how a 3D object might fracture in impact. The process works by projecting the mesh into a 2D space, scattering aforementioned seed points randomly, and then grouping the triangles of the mesh based on which seed they are closest to. While the method isn't exactly mathematically precise 3D Voronoi decomposition, it does create a visually similar result with significantly lower computational cost (Thomas & Zhang, 2022). Although this in mind the effect has been compared to glass breaking or shattering.

Mathematically speaking each seed point $\mathrm{s_i} = (\mathrm{x_i}, \mathrm{z_i})$, where $\mathrm{x_i} \in [X_{\min}, X_{\max}]$ and $\mathrm{z_i} \in [Z_{\min}, Z_{\max}]$, is randomly sampled coordinate within the object's bounding box, transposed onto the XZ plane. Each triangle in the mesh represented by vertices $v_1, v_2, v_3$, has its centroid projected into 2D: $c = \left(\frac{x_1+x_2+x_3}{3}, \frac{z_1+z_2+z_3}{3}\right)$. Then, each triangle is grouped into sets $T_k$ by its proximity to the closest seed point $S_k$ using basic 2D distance function: Set $T_k = \{c \mid \forall s_j, |\mathrm{c} - s_k| \leq |\mathrm{c} - s_j|\}$. This projection simplifies the spatial logic, allowing the created system to handle fragmentation quickly and effectively in real-time using only lightweight 2D computations.

## 3.3 Mesh Optimization & Retopology

Once the object is broken into fragments, some of the pieces may have more vertices and triangles than necessary. This can affect performance by reducing it, which will especially happen in scenes with many destructible objects as these particularly will increase draw calls. To solve this, mesh optimization is applied. This involves essentially the reduction of the number of vertices and simplifying the shape while still maintaining the general appearance of the mesh.

One approach to optimization is retopology, which, in simple terms, means rebuilding the surface of a mesh with cleaner and simpler geometry. While professional retopology involves manually placing the vertices and edges, this project uses a lightweight version of this process to create a system that can do this procedurally.

Instead of directly collapsing edges or removing small triangles, the system will procedurally generate new fragment meshes during the destruction of the old mesh. Each of these fragments is constructed by first dividing the original mesh into triangles with a Voronoi algorithm and then extending those triangles towards the center of the original mesh. This creates pyramid-like structures while also guaranteeing that each fragment is closed and has a manifold 3D shape with optimized geometry, stable normals and efficient handling of the collision.

Mathematically speaking, the center $C$ is calculated as the average of all the vertex positions in the original mesh: $C = \frac{1}{n}\sum_{k=1}^{n} v_k$ After which each triangle forms a 3D volume using the original face $\{v_1, v_2, v_3\}$ and three additional faces:$\{v_1, v_2, C\}$, $\{v_2, v_3, C\}$ and $\{v_3, v_1, C\}$. This creates a volumetric, pyramid-like fragment, ensuring that each piece is both visually coherent and physically stable as mentioned before.

## 3.4 Procedural Destruction in Unity

Procedural destruction refers to breaking or deforming objects using runtime algorithms such as scripts, rather than relying on pre-made animations or fixed models. Traditionally, Unity's destruction systems have relied heavily on physics-based simulations, which often means rigidbodies or forces or simply using pre-made broken versions of the models with ready-made fragments. While these methods do yield impressive visuals, they often require additional assets and setup time.

Procedural approaches, such as the one used in this system offer the advantage of adapting more into gameplay and user inputs. This makes the destruction effects more flexible and lighter on the resources of the development team as it depends solely on the mesh data and basic geometry math. It also makes it more reusable and easily scalable, especially in games where the focus is on ever-changing, dynamic environments.

When combined with Unity's built-in physics components—such as rigidbodies and mesh colliders—the created system is capable of handling complex destruction effects efficiently and convincingly in real time while still physically keeping consistent within the game environment.

# 4 Project Planning

For success in any development a plan is needed, the following details system's creation's planning steps to structure and guide the development procedural destruction system within Unity game engine.

## 4.1 Project System's Objective & Scope

The aim of this thesis project is to develop a system for Unity game engine which allows developers to simulate dynamic object destruction using procedural mesh fragmentation. The destruction is achieved by using a Voronoi-inspired algorithm combined with retopology-based mesh generation method. The goal is to make a system which runs efficiently in real-time and can be therefore used in applications such as games or other interactive simulations.

The focus on the project is to achieve core functionality and stable performance rather than chasing advanced visual effects or complex destruction. The resulting system is designed to be lightweight, easy to use and could be added to unity as a unity's package file without using pre-fractured models or external assets.

Key objectives of the project include:

1. Implementation of a Voronoi-style fragmentation system which would be based on grouping triangles in the mesh around specific seed points.
2. Use of center-based fragmentation method to ensure all fragments are closed manifold meshes.
3. Generating fragments of the mesh dynamically during runtime within Unity 6.
4. Achieving reasonable computational performance that would suit real-time gameplay scenarios.

## 4.2 Tools & Technologies

Unity 6 is used as the main development platform for the prototype of this project. The reasoning for this is the abundance of its properties which include its Mesh API, built-in physics engine and support for real-time scripting. With these it is possible to manipulate meshes and simulate the destruction in real-time.

As unity uses C# as its scripting language of choice for its environment the implementation of its usage also in this project is warranted. This choice allows the project easily to access and implement fragmentation logic and real-time object manipulation as well as physics interaction with Unity's built-in system.

For the writing, editing and debugging the said C# scripts Visual Studio is used as the Integrated Development Environment (IDE). This helps the project to differentiate between different classes and methods inside the script as Visual Studio color codes these.

## 4.3 Methodology

The first step involves understanding Unity's mesh structure, Voronoi diagrams and geometry processing techniques. This includes researching academic papers, technical articles and Unity's own extensive documentation to create familiarity with the topics mentioned. This is done so to guide the design of the project to be a lightweight, real-time destruction system.

The system was designed to dynamically fragment meshes by assigning triangles in the object's mesh to randomly pre-selected seed points. A center-point capping method is used to create closed fragments that are also stable.

Implementation phase involves creating a Unity C# script that could randomly generate seed points across the object's projected 2D plane, group triangles in the mesh based on their proximity with the seed points, connect triangles into

central vertex rebuilding therefore fragments. The script is also needed to add materials and convex colliders to each new fragment of the object.

After the initial implementation, mesh cleanup operations are added to ensure fragments are structurally solid. Lightweight optimization techniques are also used to simplify fragment's geometry without introducing noticeable changes to visuals. Collider generation and Rigidbody are also used to ensure performance during gameplay.

The system is also tested using simple 3D primitives such as cubes and cylinders to validate fragmentation accuracy, visual consistency and physical interactions with other fragments and the testing area. Results are thoroughly documented by using screenshots of the cubes and cylinders and with performance metrics that are found in Unity.

## 4.4 Limitations

While the system does achieve dynamic fragmentation of 3D objects, it has several limitations such as the fragmentation being based on simplified Voronoi algorithm and doesn't simulate material stress or sometimes realistic impact behavior.

The fragments generated by the system are also less realistic than some generated with fully procedural fracturing techniques as their center-based capping method is efficient but results in less realistic internal structures. All fragments are also the same material which limits visual diversity in the destruction effects.

Testing is mainly done with simple meshes, although higher poly meshes were also tested so that they would not pose threats in performance issues. Effects such as debris scattering, multi-level shattering and force propagation are also left outside the scope of the project which will result in the fragmentations being less visually impressive.

# 5 Implementation

With the system's planning and theory groundwork established, the next step is to bring the system in question into reality within Unity. The following will delve into technical implementation of the project system and explain how each part was handled inside Unity.

## 5.1 Seed Point Generation

The fragmentation process begins by projecting vertices on XZ-plane along Y-axis, enabling efficient 2D spatial reasoning during the grouping process. Then by generating a set of seed points which act as reference centers for grouping triangles. These seed points are then randomly scattered across the object's flattened 2D surface.

The process works by first retrieving the object's bounding box using Unity's MeshRenderer.bounds function. Next random positions are generated within the X and Z bounds. The number of the random points generated is determined by the user-defined parameter.

As described in 3.2, each seed point is a randomly sampled coordinate transposed onto the XZ plane using uniform distribution. These points take the form $\mathrm{s_i} = (\mathrm{x_i}, \mathrm{z_i})$, where $\mathrm{x_i} \in [X_{\min}, X_{\max}]$ and $\mathrm{z_i} \in [Z_{\min}, Z_{\max}]$. These seed points serve as centers for grouping triangles based on proximity, therefore forming the basis for the fragmentation logic.

By projecting the mesh's triangle centroids onto the XZ plane, the surface can be treated as a simplified version of a 2D domain which can be used for grouping. This way the computational cost of a full 3D Voronoi decomposition can be avoided while still offering visually coherent fragmentation. This method is also flexible as tweaking the seed point count directly changes the level of detail in the fragmentation.

## 5.2 Triangle Grouping Based on Proximity

Once the seed points are set, the object's mesh triangles are grouped based on their proximity to a seed point. This is done by calculating the centroid of each triangle by averaging its three vertex positions. The centroid is then projected onto the XZ plane.

Using 2D distance calculation, the nearest seed point is identified for each centroid. After this, triangles are assigned to their nearest seed point's group.

This grouping process directly applies the mathematical rule which was introduced in 3.2, in which each triangle centroid is assigned to the closest seed point by minimizing 2D Euclidean distance. Although the actual implementation in the created system uses Unity's own Vector2.Distance() function, it mirrors the theoretical equation: $c = \left(\frac{x_1+x_2+x_3}{3}, \frac{z_1+z_2+z_3}{3}\right)$ and $\text{Set } T_k = \{c \mid \forall s_j, |\text{c} - s_k| \leq |\text{c} - s_j|\}$

At this point, the mesh will behave as if it were a 2D UV-mapped surface. Each grouping corresponds to a region in this pseudo-UV space, even though no actual UV coordinates are used.

This approach results in contiguous triangle clusters around each seed point, forming logical fragments for the object's surface. The grouping process is quite simple and straightforward yet highly effective, striking a balance between runtime efficiency and creation of fragmentation that looks realistic enough for destruction effects.

## 5.3 Center-Based Fragment Mesh generation

After grouping the triangles by the seed points, the next step is the generation of the new fragment meshes based on those groups. This is done by copying triangles from each group into a new object and introducing a new vertex point placed at the center point of the original object.

This method follows the center-based capping strategy outlined in 3.2, where each triangle group is extended into a 3D volume by connecting its outer edges to a shared center vertex: $C = \frac{1}{n}\sum_{k=1}^{n} v_k$

To give volume to each 2D group, additional triangles are created by connecting the center point to the outer edges of the grouped triangles, forming pyramid-like shapes. Basically, each fragment extends from the central core outward through its associated triangle group, converting the 2D projection into a closed, volumetric 3D structure. In which each triangle contributes its base $\{v_1, v_2, v_3\}$ and the three faces:$\{v_1, v_2, C\}$, $\{v_2, v_3, C\}$ and $\{v_3, v_1, C\}$

This method ensures that the fragments are closed and structurally solid, making them stable for physics simulations. This way it also ensures that the generation is more efficient as it doesn't have to deal with overly intricate geometry.

After the fragments are generated their normals are recalculated using Mesh.RecalculateNormals() function. Actual UVs are also assigned based on world position, while keeping the materials consistent across fragments. This approach is heavily inspired by retopology principles discussed earlier in 3.3, as they provide clean yet efficient geometry for real-time applications.

## 5.4 Fragment Collider & Physics Setup

Finally, once the fragments are ready, they are required to behave similarly to other dynamic objects within the Unity's physics system. For each fragment a MeshCollider component is added, using the new fragment's mesh for collision detection. Said collider is set to convex to ensure proper interactions with the fragments. Finally, a component called Rigidbody is added to enable the created fragments to respond to forces and collisions in the playtime.

Fragments are positioned and rotated to match the original object's transform at the time of destruction. As the code in the scripts and in the created system is

open and public initial velocity or force impulses can be added to user's own scripts later on if needed.

As a result of this whole process after destruction each of the fragments behaves as separate objects, interacting naturally with their surroundings and adhering to Unity's physics rules.

# 6 Results & Discussion

After developing the core functionality of the project system, it is essential to evaluate how well the implementation in question will perform in practice while also keeping in mind to discuss about the findings and potential shortcomings.

## 6.1 Visual Results

To evaluate the visual quality of the project's destruction system in Unity. The fragmentation effect was applied to simple 3D objects such as cubes and cylinders as seen in Figure 1. The key focus is on how well the fragments maintained structural integrity and whether internal faces were generated correctly. The detail between seed point densities is also looked after.

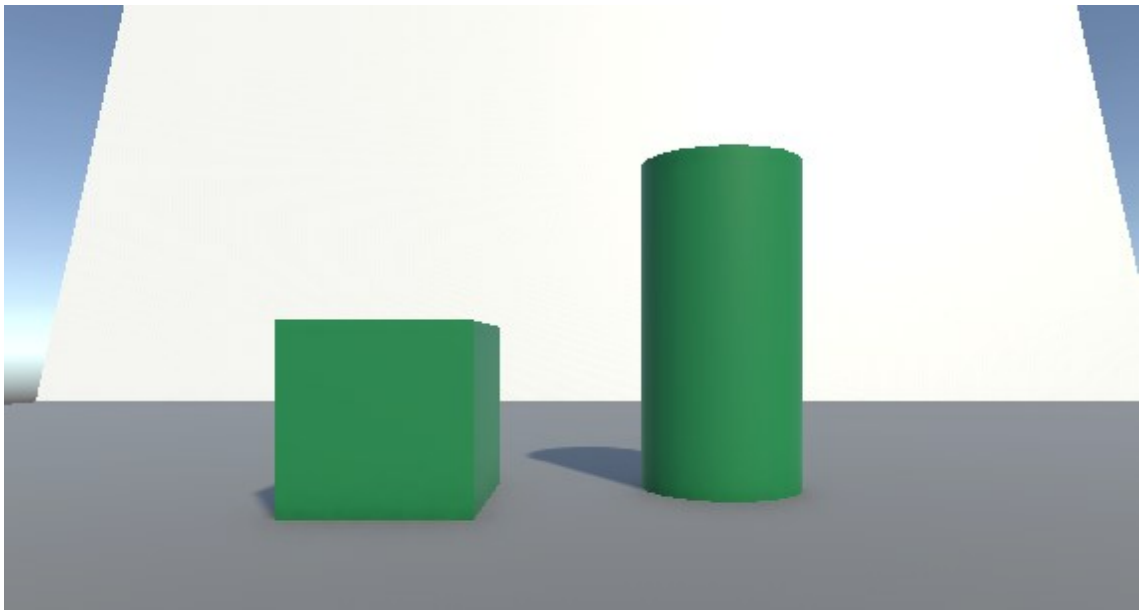

Figure 1, Cube & cylinder in the testing environment

At first, the system generated surface-only fragments by grouping triangles based on their proximity to seed points. This method produced hollow pieces with missing internal faces, as seen in Figure 2. Such hollow fragments or shards not only looked visually incomplete but also resulted in instabilities during physics simulations.

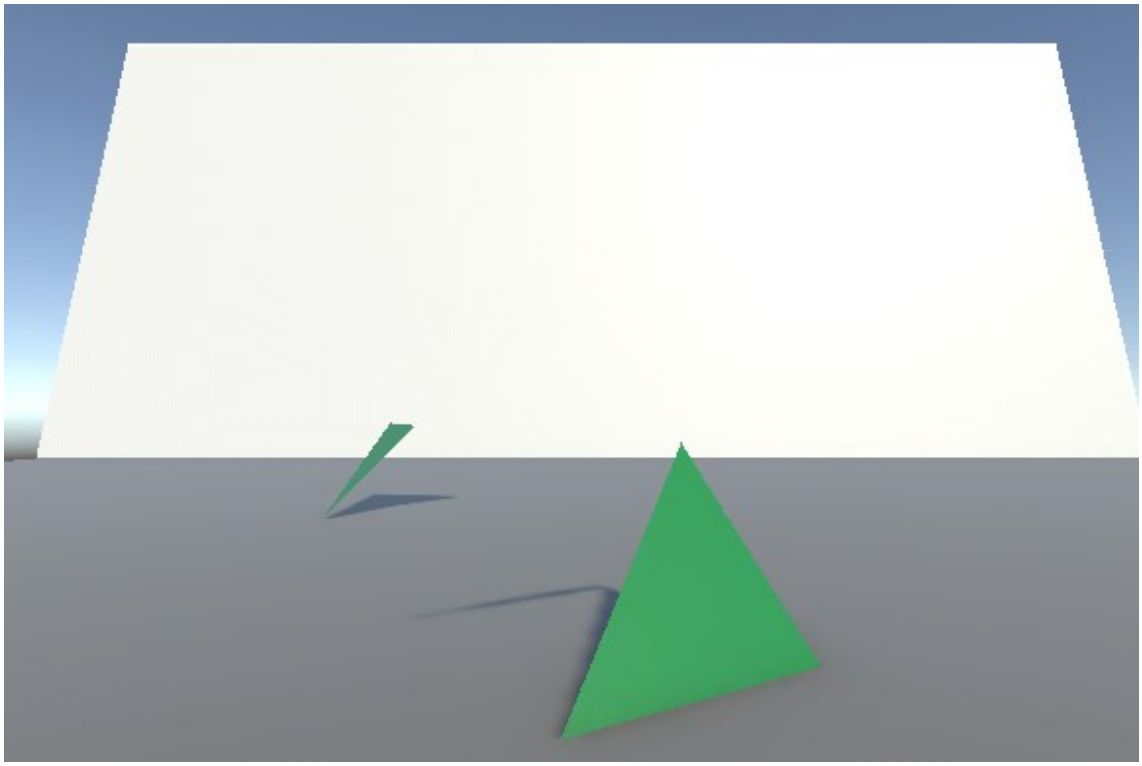

Figure 2, Cube fragments without internal faces

To improve this, the center-based extension method was added. By connecting each fragment group inward toward the original object's center, the fragments achieved their closed pyramid-like shape. As demonstrated in Figure 3, this adjustment creates fragments that are now structurally solid and visually cohesive, meanwhile also supporting the physics simulations better.

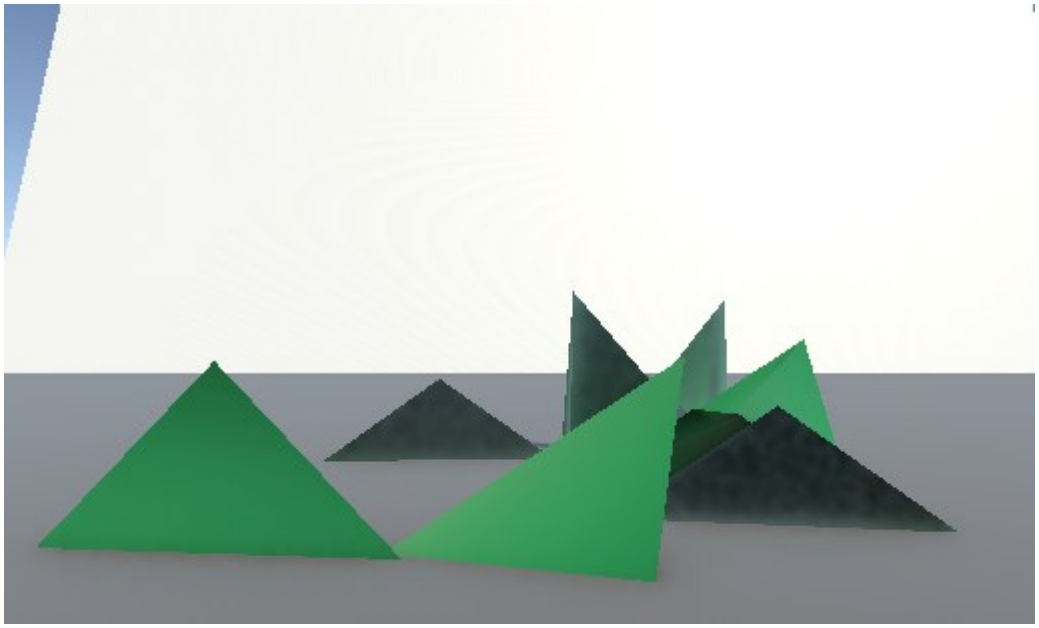

Figure 3, Cube fragments in their pyramid-like states

Another key aspect explored in this project is how the quantity of seed points affects the fragmentation detail. For this the tested object was selected to be the cylinder object. The cylinder was tested with 10, 50 and 100 seed points. With only 10 seeds the system produced larger, fewer fragments as seen in Figure 4. Increasing the seed count to 50 and 100 created more fine fragmentation pattern, as seen in Figure 5 and Figure 6.

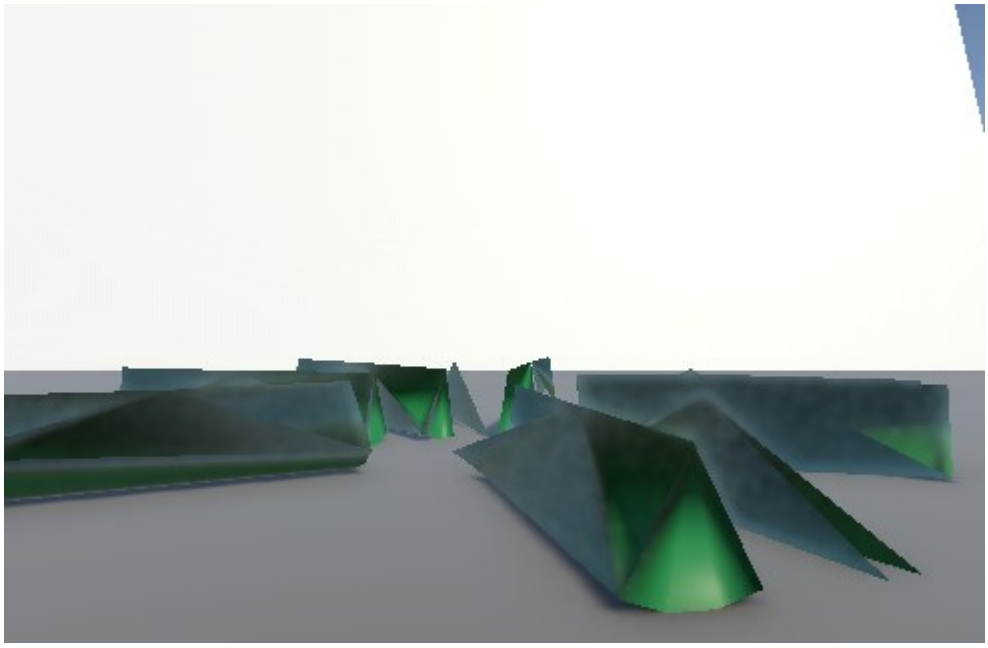

Figure 4, Cylinder object with 10 seed points

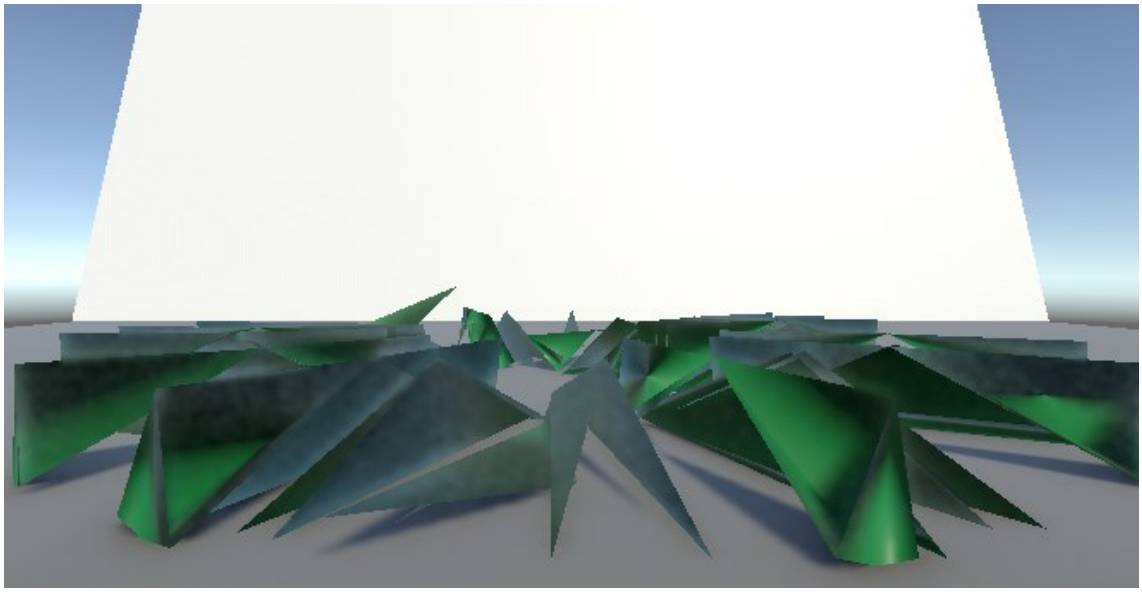

Figure 5, Cylinder object with 50 seed points

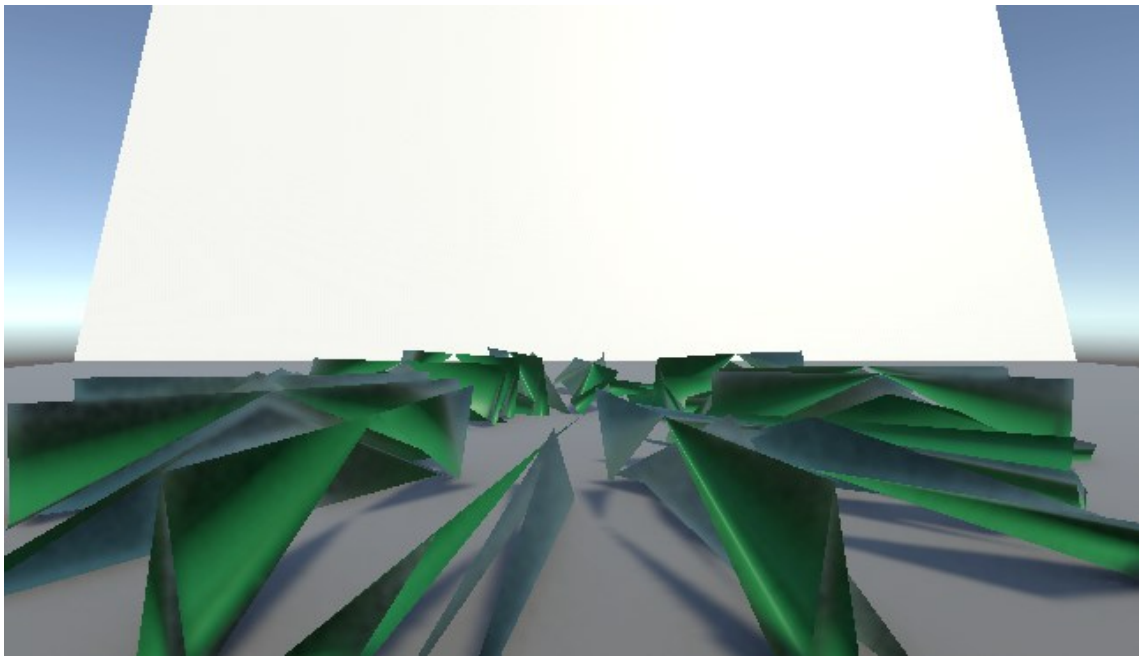

Figure 6, Cylinder object with 100 seed points

Even though the increasing the seed count slightly increases the computational cost, the boost in visual detail is significant. This flexibility allows developers to

balance performance and visual appearance depending on the current project's requirements. Although to be noted if comparing 50 and 100 seed points, the detail starts to be not noticeable. As seen in Figure 4 and 6. This may result from the vertex points of the original object having already been fully utilized during 50 seed point fragmentation, preventing the 100 seed point fragmentation from continuing.

## 6.2 Performance Evaluation

The visual test showed that the fragmentation system could consistently generate fragments that were both structurally complete as well as visually convincing across various mesh types and seed point quantities. At the same time, assessing the system's runtime performance is also important to ensure that the system is operating efficiently alongside the aesthetic appeal.

Performance testing was carried out on a simple cylinder model fragmented with different seed point counts: 10, 50 and 100 seed points. Focusing on key variables such as the number of batches, triangle (tri) and vertex (vert) counts, the number of shadow casters and any perceived impact on frame rate.

As shown in Table 1, increasing the number of seed points leads to higher batch counts and increases in the number of shadow casters, as was to be expected. Interestingly the total number of triangles and vertices remained relatively stable across different fragmentation densities, which could be showing the efficiency of the center-based fragmentation process used in the system.

Table 1, Fragmentation Performance Testing

| Number of Seed points | Batches | Triangles (tri) | Vertices (vert) | Shadow Casters | Frame rate (FPS) Impact |
|---|---|---|---|---|---|
| 10 | 53 | 6 300 | 7 000 | 24 | No noticeable difference |
| 50 | 112 | 5 200 | 7 200 | 49 | No noticeable difference |
| 100 | 189 | 5 200 | 7 400 | 84 | No noticeable difference |

Despite the rise in batches and shadow casters with greater fragmentation complexity, the real-time frame rate didn't see significant drops during the tests. These tests were, however, performed on a high-performance PC, which may have masked minor performance differences that could become more apparent with lower-end hardware.

Overall, the fragmentation system was able to maintain real-time performance even with higher fragmentation densities, demonstrating that the chosen center-based fragmentation approach and lightweight procedural generation method are suitable for real-time applications without introducing substantial computational changes.

## 6.3 Physics Behavior

Building on the visual tests, the physical behavior of the generated fragments was evaluated to ensure they interact naturally with Unity's physics system after destruction.

To achieve this each fragment was assigned a MeshCollider component set to convex and a Rigidbody component. enabling them to participate in tests for gravity, collisions and dynamic interactions. The use of the center-based extension method played a key role in ensuring that all fragments were closed, manifold meshes and provided the structural stability needed for reliable physical simulations.

During the testing, the fragments from cylinder model behaved predictably with stable falling motion and realistic collisions with both the ground and other fragments.

However, when testing with the cube model, some additional behaviors were observed. Shortly after the destruction of the cube model, many cube fragments would sometimes explode outward moving at high speeds. This was likely caused by the fragments sharing a center point which would lead to the colliders overlapping. Unity's physics engine resolves these kinds of interactions by moving the overlapping objects outwards from each other which may cause this kind of explosion to happen. These issues were not present in the cylinder tests as the surface topology is likely more varied and allows more natural distribution of the fragments' starting positions.

Despite these minor issues, the overall behavior of the fragments remained stable and consistent for both object types, especially after the initial collision events were resolved by Unity's physics engine. This indicates that the system is effective for this kind of fragment generation, though further refinements could mitigate these early-stage behaviors.

## 6.4 Limitations

While the developed procedural destruction system achieved its main objectives, testing revealed several limitations that could be addressed in future iterations.

One key limitation lies in the fragmentation method, which relies significantly on the simplified 2D Voronoi projection combined with the center-based capping. This produces visually convincing and structurally stable fragments, it doesn't account for the material stress or realistic fracture physics. As a result, fragments tend to look uniform and lack the irregularity commonly seen in physically based destruction systems.

Secondly, all fragments share the same material alignment, limiting visual differentiation between internal and external surfaces after the destruction. Future iterations could address this by implementing material blending to enhance realism.

It should also be mentioned that although the approach used in the development of the created system is inspired by the principles in the Quadric Error Metrics (QEM) algorithm for minimizing unnecessary complexity, which is a common method in 3D modelling (Garland & Heckbert, 1997). The created system does not implement QEM. Instead, it adopts the high-level goal of simplification by using custom procedural methods that are faster and suitable for real-time runtime usage.

This also makes it so that the triangles used to make the fragments in the fragmentation do not change. The center point which is used to get the pyramid shape for the fragments is then the only actually new vertex created by the system. In future this could be adjusted to create new vertices which then could also open new directions for the fragmentation complexity.

The use of Unity's MeshCollider components also introduces another constraint. Since dynamically physical objects require convex colliders fragment collision boundaries are approximated. While this ensures that the fragments are stable

physically, it sometimes results in minor inaccuracies in fragment interactions, especially for the fragments with more irregular shapes.

While testing cube models it was also observed that sometimes if the fragments shared the same center point with other fragments, this could result in fragments overlapping and Unity's physics engine resolving the conflict by rapidly pushing fragments outward, creating an explosive effect. Some cube fragments tended to be clipped into the ground plane, during initial collisions. These behaviors weren't observed with the fragments that came from cylinder objects, as in cylinder model the surface topology was more varied and allowed fragments to separate more naturally, therefore avoiding overlap.

Finally, while performance remained acceptable for smaller objects, testing on high polygon meshes as seen in Figure 7, resulted in the object's fragmentation becoming extremely small and exploding outward with even greater force leaving only small fragments behind.

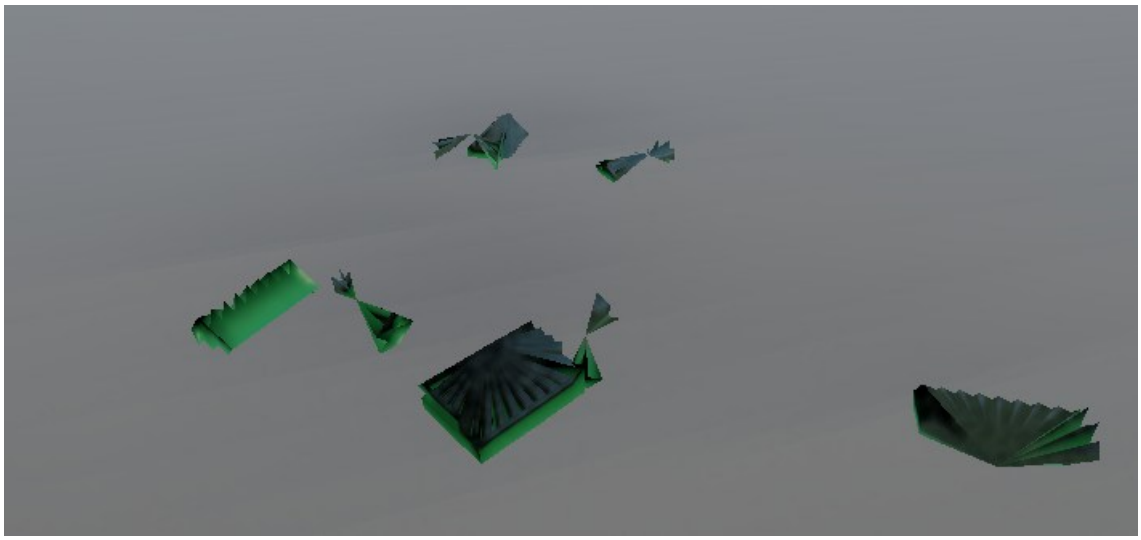

Figure 7, Fragmentation results of higher polygon mesh

Despite these limitations, the system proves to be effective for real-time use cases where lightweight and adaptable destruction effects are prioritized over complex physically accurate simulations.

# 7 Conclusion

This thesis focused on developing a lightweight procedural destruction system for the Unity game engine, designed to deliver flexible, real-time fragmentation of 3D objects without relying on pre-fractured models or heavy simulation techniques.

The system uses a simplified Voronoi-based algorithm to group mesh triangles based on their proximity to random seed points. Once grouped, a center-based extension method creates fragments that are structurally solid and enclosed. Finally, a lightweight procedural retopology approach is applied during fragment creation to ensure that the resulting pieces are efficient for real-time physics simulation while still looking cohesive.

The testing demonstrated that the system met its primary objectives:

- Fragmentation could be triggered dynamically during gameplay.
- The generated fragments were stable, manifold and integrated naturally with Unity's physics system.
- Performance remained stable with no notable frame drops even when fragmenting objects with 100 Voronoi seed points.

From a visual perspective, the system delivered convincing destruction effects. While minor quirks were observed such as fragments exploding outward due to clipping of the colliders in the cube models or occasionally clipping through the ground plane, these artifacts had no major impact on the functionality of the overall systems or realism of the destruction effect.

Although the destruction system doesn't simulate material stress or produce advanced fracture patterns, it strikes a balance between adaptability, performance and visual quality. By leveraging on simple geometric principles and Unity's built-in tools, it provides a scalable foundation that could be expanded in future work to add those missing greater realism parts and controls.

In conclusion, the system demonstrates that procedural destruction can be successfully implemented within Unity without the need to rely on external assets or complex simulations. The project system also provides developers with a practical way to achieve dynamic, runtime destruction effects for games and other interactive applications.

In future the project could be expanded by implementing dynamic material assignment to better differentiate internal and external surfaces. The system could also be modified so that its destruction is multilayered so that the fragments are destroyed again to create even more debris to better simulate the destruction happening. In summary future developments could focus on integrating more ideas into the created system to create more realistic and to ease its scalability and versality.

# References

Aurenhammer, F. (1991). *Voronoi Diagrams —A Survey of a Fundamental Geometric Data Structure.* Institute für Informationsverarbeitung Technishe Universität Graz. Retrieved from https://dl.acm.org/doi/pdf/10.1145/116873.116880

Garland, M., & Heckbert, P. S. (1997). *Surface Simplification Using Quadric Error Metrics.* Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/~./garland/Papers/quadrics.pdf

Hedén, A., & Wessman, E. (2023). *The collider tool project, Implementation of a practical tool for adjustment of complex 3D-colliders in Unity3D game engine.* Bachelor thesis, Stockholm University, Department of Computer and Systems Sciences. Retrieved from https://www.diva-portal.org/smash/get/diva2:1784460/FULLTEXT01.pdf

Incredibuild. (2025). *Unity.* Retrieved April 2025, from https://www.incredibuild.com/glossary/unity

Rey, J. (2025). *What is Unity? Game Engine Overview.* skillademia. Retrieved April 2025, from https://www.skillademia.com/blog/what-is-unity/

Ronnegren, J. (2020). *Real Time Mesh Fracturing Using 2D Voronoi Diagrams.* Bachelor thesis, Faculty of Computing, Blekinge Institute of Technology, Karlskrona, Sweden. Retrieved from https://www.diva-portal.org/smash/get/diva2:1452512/FULLTEXT02.pdf

Thomas, R., & Zhang, W. (2022). Real-time fracturing in video games. *Multimedia Tools and Applications*. Retrieved from https://link.springer.com/article/10.1007/s11042-022-13049-x

Unity Technologies. (2024). *Unity Multiplatform.* Retrieved April 2025, from https://unity.com/solutions/multiplatform

Unity Technologies. (2025, a). *Unity API.* Retrieved April 2025, from https://docs.unity3d.com/ScriptReference/index.html

Unity Technologies. (2025, b). *Unity Documentation, Mesh.* Retrieved April 2025, from https://docs.unity3d.com/ScriptReference/Mesh.html

Unity Technologies. (2025, c). *Unity Manual, Physics.* Retrieved April 2025, from https://docs.unity3d.com/Manual/PhysicsSection.html