Bachelorarbeit

Creating realistic real-time destruction using the Vienna Physics Engine

Verfasser

Andy You Wei Liu

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2026

Studienkennzahl lt. Studienblatt:	UA 033 521
Fachrichtung:	Informatik
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

1 Introduction **3**

2 Related Work **5**

3 Technical Foundation **9**
3.1 Rigid-body 9
3.2 Polytope 10
3.3 Geometry Preparation 11
3.4 Runtime Destruction 12

4 A Destruction Model for VPE **14**
4.1 Delaunay Triangulation 14
4.2 Voronoi Diagram 15
4.3 Mesh Generation 17
4.4 Fracturing Process 18
4.5 House Model 19

5 Evaluation **21**
5.1 Performance 21
5.2 Player Engagement 23

6 Conclusion **25**

1 Introduction

The roots of video game history can be found in the laboratories of computer scientists in the 1940s. Early computers were massive and expensive machines, so only large corporations and universities were able to afford them. Video games at this time were merely tech demonstration of computer capabilities and a way to gain public interest and support. It was not until the 1970s when video games truly reached the public domain [17]. The commercialization of video games also brought about the first examples of destructible environments. They could be found in arcade games such as Gun Fight (1975) and Space Invaders (1978), where players could destroy and take cover behind destructible objects. These games first introduced the concept of a dynamic destructible world that is influenced by the players action [23].

With an increase in demand for video games also came higher expectations of graphical fidelity and mechanical innovation. Allowing players to interact with and destroy objects brings a sense of realism and immersiveness to the game world [26]. The challenge behind creating such mechanisms is striking the right balance between performance and realism. The primary motivations for this thesis arises from the desire to explore, understand and replicate the principles behind destruction in gaming.

The goal of this project is to create a destruction model for the Vienna Physics Engine [15], that tries to maintain a degree of realism and accounts for the different properties and forces acting on an object during collisions. By pre-computing fractured segments that can be swapped during run-time, we can enable a seamless transition between object states. It is also important that it demonstrates reliable performance with consistent frame rates and aims to facilitate the dynamic interaction between player and the world. These metrics will be measured through qualitative and quantitative means. Performance will be evaluated based on different key performance indicators like achieved frame rates and execution times. While user interaction will be assessed through user testing, focusing on the quality of interaction between the user and the destructible environment. To ensure achieving these objectives, it is imperative to employ a systematic and iterative approach involving research and analysis of existing projects, prototyping different destruction models and user interaction testing.

When exploring destruction models in the context of games, two main research questions guide us through this project. The first one, which explores the performance of our model. Is it possible to create realistic real-time destruction in the Vienna Physics Engine? While the second one is more directed at the impact it has on player engagement. Does the introduction of a destruction model have a positive impact on player engagement towards the system? We want to understand the effects an interactive world has on players attitudes.

Different approaches exist when it comes to destruction models. The simplest and most popular paradigm is the creation of pre-factured assets. The other paradigm is to dynamically create them during run-time. The framework of our destruction model is based on the former and the principles of Voronois. After defining an amount of fractures for each object, we compute a Delaunay triangulation of said object. Given that the Delaunay triangulation is a dual graph of the Voronoi diagram, we can compute the structure of the latter. Finally, during run-time we can switch the objects with its fractured segments after a strong collision.

Subsequent to evaluating key performance indicators and assessing player feedback, we can hopefully answer our research questions and validate our hypothesis that creating

a destruction model for the Vienna Physics Engine can yield realistic destruction in real-time. Furthermore, we want to demonstrate that a destruction model positively influences player engagement with the system and that satisfaction increases when environments provide immediate visual feedback for the player's actions.

2 Related Work

This section aims to provide an overview over existing works regarding destruction models or techniques. The most popular paradigm is the making of pre-fractured assets during the development phase. In this method, artist manually or procedurally generate the necessary geometry. During destruction, the engine simply swaps the original object with its pre-generated substitutions [32].

Kotsinas and Tholén (2021) simulated the realistic destruction of 2D objects such as glass and walls using an extruding Voronoi pattern. They first determined the impact point during a collision and then generated the seed points based on this impact point, with higher density of seed points around the impact location and decreasing amounts as the distance increases. Using these seed points they generated a Voronoi diagram using the Fortune's sweep algorithm, resulting in a semi-realistic representation of an objects destruction. This approach however deteriorates in performance and visual with increasing seed points, which limits the amount of fracture patterns [20].

Ronnegren (2020) also made use of 2D Voronoi Diagram to produce fragments for the destruction of meshes. While measuring the performance of mesh generation using Voronois, they observed a linear correlation between the amount of seed points and elapsed time. For potential use in real-time application they had to limit the amount of seed points under a certain threshold. Nevertheless, they demonstrated that lower seed points can also produce realistic fragments results [38].

Fragmentation of volume meshes in three dimensions can also be achieved using a Voronoi diagram. Groenberg (2017) demonstrated that an increase in dimensionality also simultaneously incurs a significant rise in complexity. To partially circumvent this complexity, external libraries exist that can be used to calculate the Voronoi tessellation of an 3D object such as Voro++ [39] or CGAL [34]. Calculating 3D Voronois in real-time however proved to be non- viable, as the frame rate was not satisfactory enough [12].

An alternative method for partitioning 3D volume meshes involves the use of volumetric approximate decomposition. This technique breaks down complex objects, that can even be concave into smaller, simpler approximate convex meshes, which can be dynamically replaced during collision events. Furthermore, these segments can be iteratively decomposed or subjected to other additional fracturing techniques to produce better fragments [25].

Building on this concept, approximate volumetric decomposition can be achieved with Voronoi partitioning, as shown by Müller et al. (2013). The original mesh is clipped against the Voronoi decomposition to create the different convex mesh segments of an object. Afterwards, during runtime, they apply a custom fracture pattern to these segments to create the debris [32].

Another Voronoi approach utilizes a subdivision tree to store an object's fragments, similar to the method proposed by Clothier and Bailey (2015). In this structure, the root node represents the entire convex object, and as one traverses down the layers, each node is subdivided into progressively smaller parts using Voronoi partitioning. A significant advantage of this method is the capability for continuous fracturing of an object, as long as the terminal node of the tree has not yet been reached [8].

An entirely different approach to achieving repeatedly fractured objects involves allowing already broken segments to undergo further fracturing. Before this process, it is crucial to filter out objects that do not meet a specific minimum volume threshold to prevent fragmentation of objects that are already sufficiently small enough. A central

challenge in this method is determining an appropriate threshold to prevent potential issues in similar looking objects when reaching smaller volumes [43].

To enhance the visual fidelity of destructible environments, a technique known as destruction masking can be used to alter the appearance of the static geometry as demonstrated in the Frostbite 2 Engine. This process involves applying a destruction mask at the perimeter of the destroyed segment, with its spatial boundaries defined by a Signed Volume Distance Field. Additional decals can be used to further enhance the visuals of destruction [19].

While pre-creating geometry remains the industry standard due to its simplicity and minimal computational overhead, it limits the visual diversity of fragments. An alternative approach to address this limitation is to procedurally generate fragments during runtime, allowing for the creation of more realistic and varied fragment appearances [9].

The work of Van Gestel (2011) introduces the concept of destructible materials, which dictate the behavior of a material upon fracture. After establishing a library of these materials, objects can be composed using various different destructible materials, each enabling different fracture patterns. During runtime they make use of Boolean operations to generate the destruction mesh based on these fracture pattern to create the fragments [44].

Another early example of mutable environments came in the form of voxels. Voxels are the basic units of three-dimensional volumetric data and can be considered the 3D counterpart to 2D pixels. Because of their ease of modification, they are well-suited to represent non-homogeneous 3D objects and simulate their destruction by removing voxels from an object. However, voxels also come with some limitations, specifically their high memory usage, which results in a lower level of detail. This can be mitigated by using more efficient data structures (e.g., sparse voxel octrees, k-d trees) [46].

Constructive solid geometry enables the creation of complex three-dimensional objects by applying Boolean operations such as union, difference, and intersection on geometric primitives (e.g., cubes, spheres, and cones). These complex objects can be represented using binary trees, where each leaf corresponds to a geometric primitive, and each node represents a Boolean operation. For evaluating objects, one typically traverses the trees from the bottom up to combine the resulting primitives and operations. In terms of destruction, applying additional Boolean operations can simulate fracture effects, however, this approach leads to redundant geometry that requires optimization for real-time use [40].

Fracturing of brittle materials can be achieved in real-time by employing a hybrid dynamics approach that decouples the motion of an object from internal deformation. A volumetric object is discretized into a finite mesh composed of tetrahedra using the Finite Element Method. The system treats the objects as rigid bodies during motion and only computes their static equilibrium response at the discrete moment of impact. During a collision event, the internal stress tensors of tetrahedral elements are evaluated to determine if the mesh needs to be partitioned along a fracture plane [30].

Mandal et al. (2025) introduced a Galerkin-enhanced graph-based Finite Element Method designed for real-time fracturing. Unlike traditional FEM approaches that require computationally expensive re-meshing to represent separation of material, this method treats the tetrahedral mesh as a graph and projects stress along its edges. Fracture is simulated by marking edges as damaged when they reach a certain stress threshold and splitting the material along the mesh boundaries [5].

Bosch (2016) proposes a hybrid approach involving physically-based calculations and geometric partitioning to achieve impact-dependent fracturing. The method utilizes the Finite Element Method on a tetrahedral mesh to calculate the internal forces following an impact. These forces are used to determine the weights for a Weighted Centroidal Voronoi Tessellation, which dictate how an object would be split into pieces [6].

In shape matching, objects are approximated by a collection of particles, which can be independently moved by external forces. Afterwards, these scattered particles are repositioned to optimal locations that best align with the original configuration of the object. Fracturing occurs in this system when a particle exceeds a specified distance threshold from its neighboring particles in a hierarchical lattice [28].

Stegmayr (2008) explores a mesh-less numerical approach called the Element Free Galerkin (EFG) method for simulating how materials deform and fracture. Unlike traditional mesh-based methods, EFG divides an object's volume into point samples that can interact with other points within their interaction distance in a continuous field. To simulate destruction, the system uses Moving Least Squares to find the gradient of deformation, which helps determine the strain and stress in the materials [42].

Sheikh Dawood Abdul Ajees (2009) presents a solution for simulating fractures based on a probability texture. This probability texture is a 2D grayscale bitmap, where the pixel intensity correlates to the material's strength. This system checks if an impact at a specific point exceeds the material's strength. If the threshold is met, the object fractures according to this probability texture [2].

Oda and Subramaniam (2006) use a mesh refinement strategy around collision points for smaller meshes. This method involves subdividing the mesh into smaller tetrahedra within a defined space around the collision points, using the velocity and direction of the colliding object. This technique allows for a reduction in the overall number of tetrahedra while maintaining mesh quality for the fractured segments at the collision site [33].

In a mass-spring model, an object is represented as a collection of masses distributed throughout its volume. These masses are connected by springs that define the object's structural integrity. When an external force is applied, the springs stretch and compress. If the stress exceeds a predefined threshold, a spring breaks, and the stress propagates to its neighboring springs, forming cracks in the object [47].

In the point mass model, an object is represented as a set of point masses connected by rigid constraints. This model mimics the properties of brittle materials, and unlike the mass-spring model, it cannot stretch and deform before breaking. Stress exerted on each constraint is calculated using Lagrange multipliers, and crack propagation is handled in a multi-step process, allowing for natural fracture [18].

Due to the competitive nature of the video game industry, there exists a gap between academic research and industry practices concerning destruction tools. However, a recent research collaboration with Embark Studios revealed how their proprietary destruction model works. Destruction graphs use a similar approach to pre-fracturing by swapping the destruction elements during runtime. Their models consist of composite meshes that keep track of physics states, body properties, and mesh attachments. When destruction occurs, meshes can be swapped with others in a different damage state, and when the strain is too high, the connections between different meshes can be severed [10].

Unreal Engine provides a destruction system called Chaos Destruction. It is a collection of different tools that allow artists to manipulate the geometry and creation of objects. The destruction model uses an internal clustering system and geometry collection to control attachments to other geometries and define their properties. A connection

graph is used to evaluate the different strains on neighboring geometries and assess which connections can be broken [11].

The procedural destruction in Rainbow Six Siege occurs by cutting a fracture pattern out of a 3D object. A custom cut pattern is generated, influenced by the impact location and material parameters. This pattern is then clipped from a 2D projection of the object. Afterwards, the polygons are triangulated, and new 3D models are created in its place [24].

In the game Instruments of Destruction, structures are made out of blocks that are linked together, ranging from 10 to 500 pieces. Each block tracks its individual hit points based on its material and size. The game periodically checks whether the structure's layers can support its mass. If this is not possible, the structure will dynamically break apart [35].

The lead developer of Teardown, Gustafsson (2024), explained the process of optimizing his voxel destruction game by shifting from a dense voxel grid, where empty space consumes memory, to a sparse voxel format using bitmap 8x8x8 chunks. This transition resulted in a significant reduction in memory overhead. To better handle large amounts of objects, his project now features a parallel solver that uses sub-stepping [14].

Red Faction Guerrilla uses a stress system to determine when its buildings can collapse. Buildings are made up of a collection of destroyable parts, each with its own mass and strength. These parts can be grouped together to form a layer that tracks their overall mass and supporting strength. After each demolition, the engine assesses whether a layer is capable of supporting the mass situated above it. If it is deemed inadequate, the layers above will collapse [7].

Control uses the principle of granularity to represent every level of scale and detail in the game world. It also employs a destruction hierarchy, where rigid bodies break into chunks, which turn into debris, and then become dust and particles when they are damaged enough. Additionally, the game uses decals to simulate the different damage states of the geometry [37].

3 Technical Foundation

In this chapter, the fundamental concepts necessary to understand a destruction model in a physics engine will be explained. We will define how real-life objects are represented, focusing on rigid body dynamics and the logic behind collision detection using polytopes. Techniques for object fracturing will be examined, including methods to determine when an object should be fractured, utilizing concepts from Delaunay triangulation, Voronoi diagrams, and fracture mechanics. Finally, we will discuss runtime destruction strategies that can be integrated into our model to simulate realistic interactions.

3.1 Rigid-body

In physics engines, a rigid body is an idealized non-deformable solid object that maintains constant distances between any two given points. It cannot bend, stretch, or compress under physical forces and maintains its original shape. Physics engine models describe the motions of rigid bodies through Newton's Three Laws of Motion [31]:

1. A body at rest remains at rest, and a body in motion continues in a straight line at constant speed unless acted on by a external force.
2. Alteration of velocity is proportional to the external force applied and occurs in the direction of that force.
3. When two bodies interact, the forces they exert on each other are equal in magnitude and opposite in direction. For every action there is an equal and opposite reaction.

A rigid body's position $p(t)$ can be represented by a three-dimensional vector that specifies the location of the rigid body's center of mass in world space. The center of mass in rigid bodies is the average location of the bodies mass distribution. This concept is important, as all linear and angular velocities are applied at this point and serve as the origin of the body's local coordinate system [31].

The orientation of a rigid body $R(t)$ can be represented using a 3x3 rotation matrix. However, the downside of using rotation matrices is the excess degrees of freedom and the need to re-orthogonalize them to ensure valid rotation. Alternatively, quaternions [4] provide a more efficient representation of a body's orientation. A quaternion extends complex numbers and utilize four scalar values, which reduce computational complexity, avoid gimbal lock and save on interpolation of rotations [31].

The derivative of position over time is the linear velocity of a body, expressed as $v(t) = \dot{r}(t)$. It characterizes how the center of mass of the body is moving through space. Furthermore, the derivation of velocity is the acceleration of a body, denoted as $a(t) = \dot{v}(t)$, which describes how the velocity changes over time. Both linear velocity and acceleration can be represented as a three-dimensional vector, with the axis reflecting the direction and the magnitude corresponding to speed. In addition to linear motion the time derivative of orientation describes the angular velocity. This describes the rotation of a body around an axis that passes through the center of mass. The angular velocity $\omega(t)$, is also represented as a three-dimensional vector, with its direction indicating the axis of rotation and the magnitude representing the speed of rotation around this axis. To manipulate the angular velocity, a rotational force known as torque τ must be applied to the body. The relationship between torque, inertia tensor and acceleration is given by the equation $\tau = I \cdot \alpha$, where I is the inertia tensor and α is the angular acceleration [31].

A rigid body can be characterized by the region its mass lives in, which can be conceptualized as a collection of particles distributed throughout this volume. The total mass of the rigid body is calculated as the sum of mass of each individual particle as seen in Equation (1) [31].

$$M = \sum_{i=1}^{N} m_i \tag{1}$$

The linear momentum $P(t)$ of a rigid body is the sum of momentum of each individual particle within the body. The time derivative of the linear momentum corresponds to the total external forces $F(t)$ acting on the body. Conversely, the angular momentum $L(t)$ is expressed as the product of the inertia tensor and the angular velocity. The torque $\tau(t)$ can be determined by the time derivative of the angular momentum. The inertia tensor characterizes the distribution of mass within a body and quantifies its resistance to rotational motion around the different axes as seen in Equation (2). The leading diagonals I_{xx}, I_{yy}, I_{zz} represent the moments of inertia about each principal axes. The remaining entries are the products of inertia [31].

$$I(t) = \sum_{i=1}^{N} \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix} \tag{2}$$

In a physics simulation, the complete state of a rigid body can be represented by a state vector $Y(t)$ compromised of four elements: position, rotation, linear momentum and angular momentum. Using the derivative of this state vector we can get additional information, such as linear velocity, angular velocity, total forces and torque as seen in Equation (3). Throughout the simulation the mass and inertia tensor of a body remain constant and are defined beforehand [31].

$$Y(t) = \begin{pmatrix} p(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} \quad \dot{Y}(t) = \begin{pmatrix} v(t) \\ \omega(t) \cdot R(t) \\ F(t) \\ \tau(t) \end{pmatrix} \tag{3}$$

3.2 Polytope

In a physics engine, a polytope functions as the collision geometry attached to rigid bodies. While the rigid body represents dynamic properties, it is essential to attach a geometric shape that defines the space the body occupies for accurate collision detection and response. The Vienna Physics Engine restricts collision-capable bodies to convex collider shapes because they enable efficient and numerically stable collisions. A polytope is fundamentally defined as a geometric object with flat sides and are a generalization of three-dimensional polyhedra to any number of dimensions. It can be characterized as the convex hull of a finite set of points in n-dimensional Euclidean space, representing the smallest convex shape that encompasses a collection of points. There are two mathematically equivalent definitions of a convex polytope [29]:

- V-Polytope: This is defined as the set of all convex combinations of a finite set of points, effectively creating the convex hull of these points.

- H-Polytope: This is defined as the intersection of a finite number of closed linear half-spaces.

Physics engines predominately use the vertex representation (V-Polytope) for defining collision shapes and facilitating collision detection. In a rigorous topological framework, a polytope comprises elements of varying dimensionality organized within a hierarchical structure. These elements are referred to as j-faces, where j denotes the dimension of the element [13]:

- 0-faces: Known as vertices, these are single points in space
- 1-faces: Known as edges, these are line segments connecting two vertices
- 2-faces: Known as faces, these are polygons that define the exterior boundary of the object in three-dimensional space

Collision detection typically occurs in three distinct phases. In the broad phase, all possible groups of objects are filtered using coarse bounding volumes or spatial data structures. This step helps to determine which objects cannot possibly collide, significantly reducing the number of potential interactions that need to be examined. In the optional middle phase, objects are fitted with more tighter bounding volumes to refine the accuracy of collision detection. Finally, in the narrow phase, the remaining object pairs are checked for collisions. During this phase, a contact set is generated, providing the contact normal, which specifies the direction along which the collision occurred, the contact points, the specific locations in where two object are touching and how far the objects are overlapped. Using this information, the physics simulation can infer how objects should interact with each other and which impulse to apply [1].

3.3 Geometry Preparation

A destruction model defines when an object breaks, how fragments are generated and how they interact within the physics simulation. Most modern physics engines separate destruction into two distinct stages. The first is the preparation of geometric data. In this step, artists or tools analyze and subdivide a model into its fragmented segments, which is later used when the object breaks apart. This involves using algorithms or content creation tools to achieve but there are also techniques to achieve this during runtime. The second stage involves triggering the destruction and handling the fracturing process of an object during runtime.

Pre-fracture and procedural approaches represent two distinct methodologies in geometry preparation. The pre-fracture approach involves creating geometric assets based on predefined templates. In contrast, the procedural approach emphasizes the use of algorithms and procedural generation techniques to create geometry dynamically, producing more unique and complex geometric forms.

There are many ways to prepare geometry for fracturing purposes. The most prevalent technique used comes in the form of Voronoi diagrams. Voronoi diagram are extensively applied in different area of computer graphics. A Voronoi diagram is the partition of space from a finite set of seed points. Each seed point creates a convex region, also known as the Voronoi cell. The area enclosed in this cell are defined as the points that are closer to this seed point than to any other seed point [21].

Delaunay triangulation is also a fundamental geometric concept utilized to partition a set of points into a mesh of triangles. The core defining property of Delaunay triangulation is the empty circumsphere criterion, which stipulates that the circumcircle of every triangle must not contain any other points from the set within its interior. This criterion ensures a uniquely defined mesh for points in general position and also establishes a mathematical duality with the Voronoi tessellation [22].

Among the various methods for constructing Delaunay triangulations, incremental point insertion algorithms, such as the Bowyer-Watson algorithm [27], are particularly popular. This algorithm typically begins by computing a supertriangle that encloses all points intended for triangulation. For each new point introduced, the algorithm identifies all existing triangles whose circumcircles include that point. Once these triangles, failing the circumcircle test are captured, they are removed, creating a concavity within the mesh. The algorithm then completes the update by creating triangles that connect the new point to every edge situated on the boundary of the newly formed cavity. Because it operates incrementally, the algorithm ensures that the triangulation remains a valid Delaunay triangulation after each point insertion [36]. The Voronoi diagram and Delaunay triangulation are mathematically dual structures, meaning that one can be derived from the other. To compute a Voronoi diagram, the dual relationship can be utilized as follows in 2D [21]:

- A Delaunay vertex corresponds to a Voronoi cell
- A Delaunay edge corresponds to a Voronoi edge
- A Delaunay triangle corresponds to a Voronoi vertex

After preparing the geometry, it can be effectively utilized during runtime destruction scenarios. However, there are alternative approaches that facilitate real-time geometry preparation influenced by various factors occurring during runtime. For instance, in the event of a collision, the impact point can serve as the focal point for distributing seed points for the Delaunay triangulation. This method generates a denser concentration of points around the impact point, tapering off to lower density as the distance from the center increases. Such a distribution leads to a more dynamically fractured-looking object [9].

3.4 Runtime Destruction

In addition to generating the necessary geometry, a destruction model must also determine the specific scenarios in which destruction should occur. In fracture mechanics, all structures within a material contain inherent flaws, and the failure of these structures is governed by the behavior of these flaws. Several internal and external factors influence that failure. For our purposes, we focus on the concept of Fracture Toughness, which represents the critical stress threshold that a material can withstand before fracturing occurs [3]. To apply this concept, we simplified the characterization of fracture mechanics to involve impulses and force thresholds.

In physics engines, collisions are commonly resolved using impulses rather than continuous forces. This likewise applies to the Vienna Physics Engine. Therefore, we can consider impulses to determine whether a fracture should occur during collisions between two objects. In classical mechanics, impulse J is defined as the change in momentum of an object, which can be mathematically expressed in Equation (4):

$$J = \Delta p \tag{4}$$

$$p = m \cdot v \tag{5}$$

where momentum p, as seen in Equation (5), is the product of an object's mass m and its velocity v [41].

4 A Destruction Model for VPE

The destruction model determines the destruction logic and creates the necessary geometry for the destruction process. The rigid body physics simulation is provided by the Vienna Physics Engine [15], with rendering handled by the Vienna Vulkan Engine [16]. Fragmentation is produced by Voronoi tessellation of the original polytope. To generate that tessellation efficiently we first compute the Delaunay triangulation and then convert it to the corresponding Voronoi diagram. Afterwards we extrude the geometry into 3D and generate the corresponding meshes and OBJ files.

4.1 Delaunay Triangulation

To compute the triangulation of a polytope, it is essential to first determine its bounding box and create a 2D projection onto the xy-plane. This process involves iterating through the vertices of the polytope to identify and record the minimum and maximum values for the x, y and z dimension. Once this information is gathered, we focus on selecting the vertices that correspond to the highest z-dimension. These vertices will be later used for seed point generation and clipping purposes.

With the xy-coordinate values established, we can generate a specified number of random points within the bounding box defined by the 2D projection. To ensure the robustness of the triangulation, any duplicate points will be removed to prevent degenerate edge cases from influencing the results. These unique points will then serve as the seed points for the Delaunay triangulation. Utilizing the Bowyer-Watson algorithm, demonstrated in Listing 1, we can effectively compute the Delaunay triangulation of the polytope.

```
1  function BowyerWatson (pointList)
2      // pointList is a set of coordinates defining the points to be
       triangulated
3      triangulation := empty triangle mesh data structure
4      add super-triangle to triangulation // must be large enough to
       completely contain all the points in pointList
5      for each point in pointList do // add all the points one at a time
       to the triangulation
6          badTriangles := empty set
7          for each triangle in triangulation do // first find all the
       triangles that are no longer valid due to the insertion
8              if point is inside circumcircle of triangle
9                  add triangle to badTriangles
10         polygon := empty set
11         for each triangle in badTriangles do // find the boundary of the
        polygonal hole
12             for each edge in triangle do
13                 if edge is not shared by any other triangles in
       badTriangles
14                     add edge to polygon
15         for each triangle in badTriangles do // remove them from the
       data structure
16             remove triangle from triangulation
17         for each edge in polygon do // re-triangulate the polygonal hole
18             newTri := form a triangle from edge to point
19             add newTri to triangulation
20     for each triangle in triangulation // done inserting points, now
       clean up
```

```
21          if triangle contains a vertex from original super-triangle
22              remove triangle from triangulation
23      return triangulation
```

Listing 1: Wikipedia Bowyer Watson Pseudocode [45]

To calculate the super triangle, we need to know how the seed points are distributed in space. We can use the midpoints of the minimum and maximum of x and y as our starting values for the corner points and translate them by a scalar times deltaMax, which is the biggest range between x or y. This ensures that the triangles is large enough to encompass all points as seen in Listing 2.

```
1     Triangle({mid_x, mid_y + deltaMax * scale},
2             {mid_x - deltaMax * scale, mid_y - deltaMax},
3             {mid_x + deltaMax * scale, mid_y - deltaMax});
```

Listing 2: Super Triangle

The Bowyer-Watson Algorithm is an incremental algorithm, that deletes invalid Delaunay triangle after each iteration. Therefore we need to keep track of which triangles and edges to delete. This was done using boolean values for a Triangle and Edge class. We decided to keep track of the triangles and edges separately, to avoid needing to compare triangles with each other. At the end of each iteration we delete all triangles that contain the current point and all duplicate edges. With the remaining edges we can form new triangles using the current point in the concavity left behind.

4.2 Voronoi Diagram

Upon completion of the Delaunay triangulation, we can transition to constructing the dual Voronoi tessellation. Since the Delaunay vertex serves as the center point of its corresponding Voronoi cell, we can efficiently re-utilize the seed points from the Delaunay triangulation and iterate through them as seen in Listing 3.

```
1  function transformToVoronoi(delaunayTriangulation, pointList)
2      voronois := emptyList
3      for each point in pointList
4          voronoi := new voronoi
5          triangleEdges := emptyList
6          for each triangle in delaunayTriangulation
7              if triangle contains point
8                  add edges incident to point to triangleEdges
9          for each edge in triangleEdges
10             newEdge := form dual of edge
11             add newEdge to voronoi
12     return voronois
```

Listing 3: Delaunay Triangulation to Voronoi Diagram Pseudocode

For each seed point, we find all triangles that include that point as a vertex, then collect the triangle edges that touch the seed point. We store this mapping in a hash map where the key is an edge (Edge class) and the value is a vector of triangles that contain that edge. Concretely, for every triangle that has the seed point as one of its vertices, we examine the triangle's three edges and add the two edges that include the seed point to the hash map vector entry for that edge.

Subsequently, to compute the vertices of the Voronoi diagram we save the circumcenter of adjacent triangles that share an edge with the seed point. In situations where only a

single triangle exists for an edge, it is essential to compute the perpendicular bisector of that edge. These scenarios typically arise in Voronoi cells that are unbounded. For these cases, we calculate the perpendicular line segments for the Delaunay edge, anchor them at the circumcircle of the Delaunay triangle, extend them for a defined distance and then connect them, effectively creating bounded edges as seen in Listing 4. This adjustment ensures the Voronoi diagrams remains closed for subsequent clipping operations. The Vienna Physics Engine employs a left-handed coordinate system. Therefore the vertices are saved in a vector in counterclockwise order in a Voronoi class as shown in Listing 5.

```
glmvec2 A = edge.getA();
glmvec2 B = edge.getB();
glmvec2 C = triangle[0].getCircumcircleCenter();
glmvec2 D = intersectionOnLineFromPoint(A, B, C);

glmvec2 DC = C - D;

glmvec2 P1 = C + glm::normalize(DC) * 10.0_real;
glmvec2 P2 = C - glm::normalize(DC) * 10.0_real;

/**Check orientation of edge, should go away from the voronoi */
glm::length(P1) < glm::length(P2) ?
unbounded_edges.push_back(Edge(C, P2)) :
unbounded_edges.push_back(Edge(C, P1));
```

Listing 4: Unbounded edges

```
inline static bool angle_comparison(const glmvec2 &p1, const glmvec2 &p2
    , const glmvec2 &center)
{
    double angle1 = std::atan2(p1.y - center.y, p1.x - center.x);
    double angle2 = std::atan2(p2.y - center.y, p2.x - center.x);
    return angle1 < angle2;
}

inline void sort_vertices_ccw(std::vector<glmvec2> &unordered_vertices)
{
    glmvec2 center = getAveragePoint(unordered_vertices);
    std::sort(unordered_vertices.begin(), unordered_vertices.end(),
    [&center] (const glmvec2 &p1, const glmvec2 &p2)
    {
    return angle_comparison(p1, p2, center);
    });
    auto duplicates = std::unique(unordered_vertices.begin(),
   unordered_vertices.end());
    unordered_vertices.erase(duplicates, unordered_vertices.end());
}
```

Listing 5: Sorting vertices in counterclockwise order

Clipping is essential to ensure that the Voronoi cells remain confined within the boundaries of the 2D projection of our polytope. For this clipping operation, we first have to bring the vertices in a clockwise order. We can simply reverse this order by using `std::reverse()` on the vertex vector. To achieve clipping, we apply the Sutherland-Hodgman algorithm, that is already defined in the Vienna Physics Engine for collision detection purposes, to clip each Voronoi cell against the 2D bounding box defined by the original polytope.

Once the clipping process is complete, we translate each Voronoi cell to the origin, as they still remain within the coordinate space of the original polytope. This translation is accomplished by calculating the geometric center of the Voronoi cell and subtracting each vertex with this point. We also need to save this local translation of each fragment, so we that can later translate them back to their original position within the polytope after fracturing. With the Voronoi cell positioned at the origin, we can then extrude it into the three-dimensional space, thereby facilitating the computation of the corresponding polytope structures and their OBJ files.

4.3 Mesh Generation

The Vienna Physics Engine represents the geometry of objects using a polytope class, while the Vienna Vulkan Engine visualizes their 3D geometry with Wavefront .obj files. The OBJ format encodes vertex positions, texture coordinates (UVs), vertex normals, and polygonal faces that reference those vertices, UVs, and normals. To simulate and render each fragment, we must generate a matching polytope and its OBJ file.

Because we extrude Voronoi cells, the vertex positions of each polytope are already known. Voronoi vertices are sorted counterclockwise, so edges follow directly by connecting neighboring vertices. The front and back faces are formed from the vertices at the higher and lower z coordinates, respectively. Side faces are created by pairing each edge on the front face with the corresponding edge on the back face (i.e. connecting two adjacent front-face vertices to their matching back-face vertices) as seen in Listing 6.

```
for (int index = 0; index < vertices_size; ++index)
{
    if (index == vertices_size - 1)
    {
        faces.push_back({
        {index + vertices_size * 2, 1},
        {index + vertices_size, 1},
        {index + vertices_size + 1, -1}, {index, -1}});
    }
    else
    {
        faces.push_back({
        {index + vertices_size * 2, 1},
        {index + vertices_size, 1},
        {index + vertices_size * 2 + 1, -1},
        {index, -1}});
    }
}
```

Listing 6: Creating side faces for polytope

The OBJ generation follows the same overall approach but requires additional triangulation of faces. To accomplish this, we add a central vertex for the front face and the back face, then triangulate each face around its center. For the side faces we split each rectangular face into two triangle faces with one connecting diagonal. Additionally, we compute per-vertex normals and texture coordinates. For normals, we can compute the face normal for the front, back and side faces using the normalized cross product from one of their triangle using any three vertices and assign that normal to their respective face for shading. For simplicity, we used solid color for each face, meaning we can reuse the same triangle UV coordinates for every face.

4.4 Fracturing Process

For destruction to occur, each destructible body is equipped with a collision callback function in the physics engine. The physics engine periodically calculates the states of objects within the simulation and triggers this function whenever a collision involves the body as shown in Listing 7. Upon triggering, the function retrieves data from both colliding bodies. Fracturing of the polytope occurs when a specific impulse threshold is exceeded. This impulse is calculated by considering the bodies mass and velocity of the fracturing object, as well as the coefficient of restitution, which determines the elasticity of the body. Additionally, if the force exerted by the other colliding object surpasses this limit, fracturing can also occur.

```
VPEWorld::callback_collide onCollideFracture =
  [this](std::shared_ptr<VPEWorld::Body> body1, std::shared_ptr<VPEWorld
    ::Body> body2)
{
  real impulse1 = (1 / body1->m_mass_inv * glm::length(body1->
   m_linear_velocityW)) * (1 - body1->m_restitution);
  if (impulse1 > fracture_force_threshold)
  {
    std::string name = body1->m_name;
    glmvec3 position = body1->m_positionW;
    glmquat orientation = body1->m_orientationLW;
    glmvec3 velocity = body1->m_linear_velocityW;
    glmvec3 angular_velocity = body1->m_angular_velocityW;
    m_physics->eraseBody(body1);
    fracture(name, position, orientation, velocity, angular_velocity);
  }
  else if (body2->m_name != "Ground")
  {
    real impulse2 = (1 / body2->m_mass_inv * glm::length(body2->
   m_linear_velocityW));
    if (impulse2 > fracture_force_threshold)
    {
      std::string name = body1->m_name;
      glmvec3 position = body1->m_positionW;
      glmquat orientation = body1->m_orientationLW;
      glmvec3 velocity = body1->m_linear_velocityW;
      glmvec3 angular_velocity = body1->m_angular_velocityW;
      m_physics->eraseBody(body1);
      fracture(name, position, orientation, velocity, angular_velocity);
    }
  }
};
```

Listing 7: Callback collide function

When executing the fracturing process, the original body's state, its position, orientation, linear velocity and angular velocity is saved. The original body is then deleted and replaced with its fractured segments. To differentiate the different body types we used a string identifier to determine with which segments to replace the body with. The fragmented segments are spawned at the original position offset by their local position within the polytopes local coordinate space. Each segment is initialized with the original orientation and velocities of the original polytope as shown in Listing 8.

```
if (name.find("Wall") != std::string::npos)
{
```

```cpp
  for (int index = 0; index < wall_poly.size(); ++index)
  {
    VESceneNode *cube0;
    VECHECKPOINTER(cube0 = getSceneManagerPointer()->loadModel(
        "The Cube" + std::to_string(m_physics->m_body_id),
        "../../media/models/test/destruction",
        "wall" + std::to_string(index) + ".obj",
        0,
        getRoot()));

    auto body = std::make_shared<VPEWorld::Body>(
        m_physics,
        "Body" + std::to_string(m_physics->m_bodies.size()),
        cube0,
        &wall_poly[index],
        glmvec3{1.0_real},
        position + orientation * wall_tl[index],
        orientation,
        velocity,
        angular_velocity,
        1.0_real / 100.0_real,
        m_physics->m_restitution,
        m_physics->m_friction);

    body->setForce(0ul, VPEWorld::Force{{0, m_physics->c_gravity, 0}});
    body->m_on_move = onMove;
    body->m_on_erase = onErase;
    m_physics->addBody(body);
  }
}
```

Listing 8: Fracturing method

4.5 House Model

To demonstrate the destruction simulation, we constructed a simplified house model composed of destructible rigid bodies with varying sizes and shapes. Because fragment count depends on the number of Voronoi seed points 1, we adjusted the seed density per polytope, larger polytopes required more seed points to produce fragments comparable in size to those in smaller polytopes. Conversely, smaller polytopes achieved realistic fragment sizes with fewer seed points.

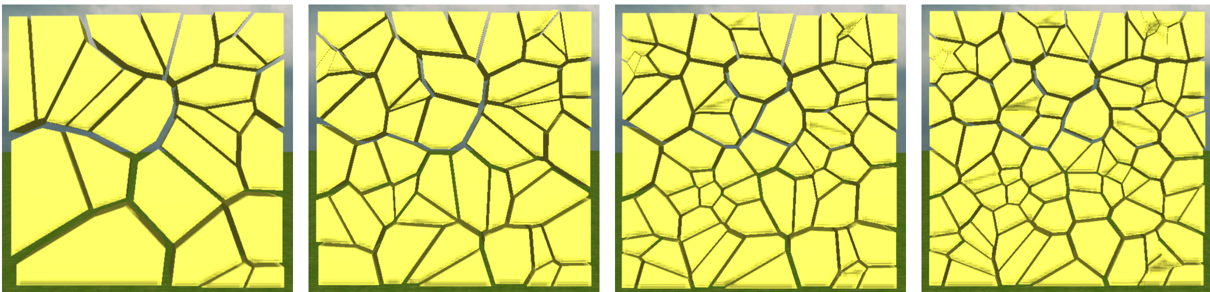

Figure 1: Voronoi diagram with 25, 50, 75 and 100 seed points

The goal of this project was to simulate destruction in real time. Using the Vienna Physics Engine, we subjected the house model to dynamic impacts by launching rigid

boxes at it and observing the resulting damage. Owing to computational and stability constraints, the fracture amount was kept at a conservative level to maintain real-time performance. A representative result of the simulation is shown in Figure 2.

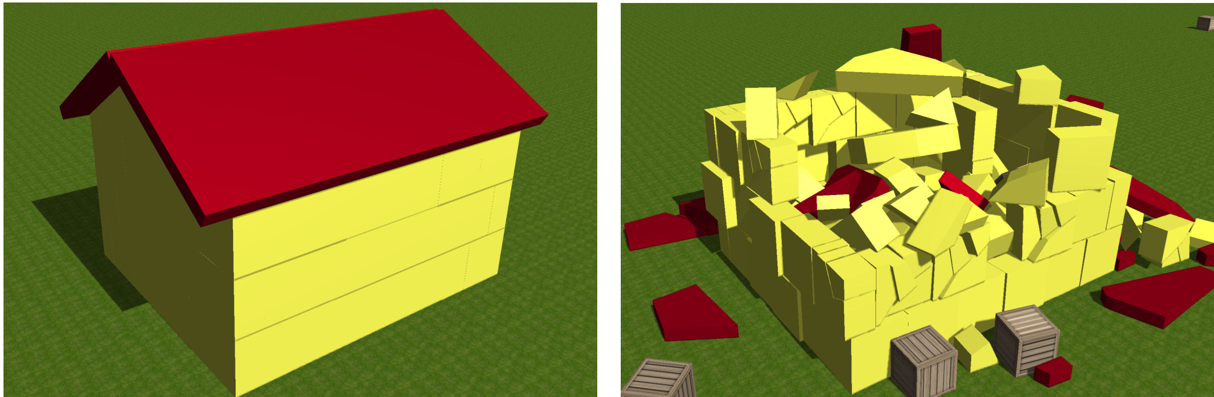

Figure 2: House before and after Destruction

The building model is composed of three polytope types: two rectangular prisms and one triangular prism. The dimensions of the polytopes are as follows:

- Yellow wall (rectangular prism): 1.00 m (length), 1.00 m (width) and 0.50 m (depth).
- Yellow Gable wall (isosceles triangular prism): 5.00 m (base), 2.89 m (equal sides) and 0.50 m (depth).
- Red roof (rectangular prism): 3.50 m (length), 3.00 m (width) and 0.25 m (depth).

5 Evaluation

In this section, we evaluate our destruction model using two distinct methodologies. The first method involves measuring key performance indicators of our model, while the second method engages participants to evaluate our solutions. By analyzing the results of both approaches, we aim to address our research questions and discuss the implications of our findings.

5.1 Performance

Component	Name
CPU	AMD Ryzen AI 9 HX 370
RAM	16 GB LPDDR5X-7500
GPU	RTX 4070 Laptop GPU

Table 1: Testing Hardware

The project was implemented and tested on a laptop running Windows 11 with following hardware specifications seen in Table 1. To assess the suitability of our solution for real-time destruction, we measured the execution time of Voronoi diagram generation, the execution time of polytope swapping after fracturing and the frame per seconds of the simulation with increasing bodies. Time measurements were done using the `std::chrono` library in C++. We established time points before and after function processes to accurately capture the execution duration as demonstrated in Listing 9. By computing the difference between these two time points, we obtained the total time taken for the different processes. The frame rate data was obtained using the integrated frame counter within the Vienna Physics Engine.

```
auto start = std::chrono::high_resolution_clock::now();
auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
    end - start);
```

Listing 9: exeuction time

We developed a PowerShell script to iterate through 1 to 100 seed points for Voronoi generation, recording the execution duration and saving it in a text file. This script was repeated ten times and afterwards we computed the average execution time for each seed count. These averages were subsequently plotted using Python in a Jupyter notebook for visualization. Our x-axis is the fragment count for the Voronoi diagram, whereas the y-axis is the elapsed time in milliseconds. The results indicate an approximately linear relationship between the number of fragments and the computation time, as illustrated in Figure 3. For 20 fragments it takes roughly under 5 ms, 40 takes around 8 ms, 60 takes around 12 ms, 80 around 16 ms and 100 takes roughly 20 ms. Given a minimum target of 60 frames per second, our implementation must complete Voronoi generation within 16.67 milliseconds, which restricts the fragment count to approximately under 80 fragments for real-time use.

While an increase in the number of fragments, as illustrated in Figure 1, does not necessarily correlate with more realistic fracture simulations, we can assume that a smaller number of fragments can still yield convincing results. The time data for polytope swapping was collected manually in increments of 10 fragments. This process was also repeated

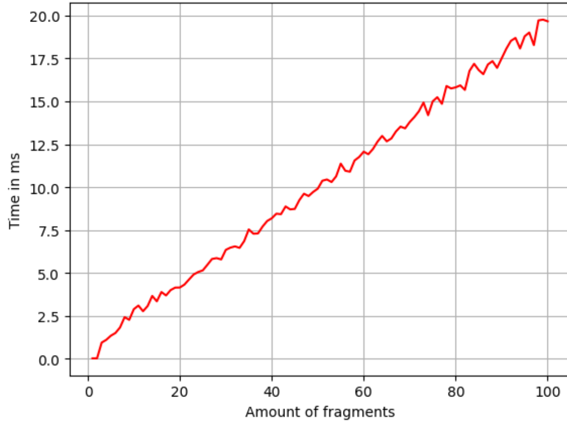

Figure 3: Execution time of Voronoi generation

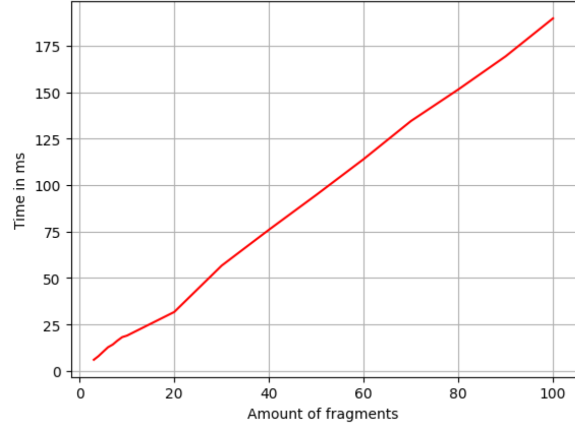

Figure 4: Execution time of polytope swapping

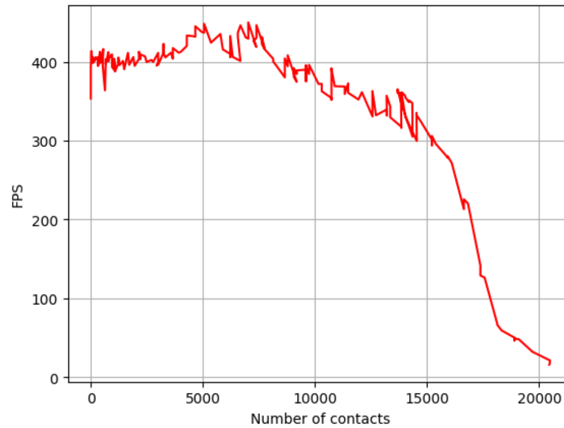

Figure 5: FPS per number of contacts

10 times and was visualized using Python as seen in Figure 4. The graph shows the amount of fragments on the x-axis and the time taken in the y-axis. We found out that swapping a single polytope takes approximately 1 to 2 milliseconds and increases linearly with an increasing amount of polytope fragments. For real-time application fragment count under 20 would be preferable, however because swapping occurs sequentially, higher fragment counts till 30 are also unnoticeable.

We also examined how frames per second (FPS) varies based on the number of contacts between bodies in the physics simulation, as shown in Figure 5. To do this we threw an increasing amount of destructible bodies on a pile of polytopes and tracked the fps using the integrated FPS counter in the Vienna Physics Engine. The figure indicates that the simulation begins to stutter when the number of contacts rises above 20,000. However, the measurements may include potential discrepancies, as the FPS exhibited a noticeable disconnect between readings and human observational assessments.

Our house model consists of 77 polytopes, each utilizing fewer than 10 fragments. Upon spawning the house model, we observe around 3,000 contacts, significantly below the threshold for real-time FPS as indicated in Figure 5. This demonstrates that our destruction model can effectively simulate destruction in real-time. The lower fragment count in our house model not only ensures efficient performance but also provides a time buffer that facilitates potential implementation of additional destruction mechanics. For

example, allowing for potential procedural generation of fragments based on the impact point.

5.2 Player Engagement

Apart from objective performance metrics, we evaluated player perception and engagement with the destruction model using an inferential questionnaire designed to probe participants views on the models performance, realism, and overall satisfaction. The survey showcased two videos of different models, one with destruction enabled and on without. In these videos, a house model is thrown at with rigid boxes and the simulated physics were displayed. Afterwards participants had to fill out the survey, which was comprised of eight statements rated on a four-point Likert scale (Strongly disagree = 0, Disagree = 1, Agree = 2, Strongly agree = 3). Responses were analyzed using inferential statistics and reported with p-values for hypothesis tests to determine whether observed attitudes differed significantly from chance. The survey was completed by n = 10 participants. Sample demographics were 90% male and 10% female, all participants were from Austria, and participants ages ranged from 20 to 30 years. Because of the small sample size, inferential results are presented with caution and are accompanied by p-values to aid interpretation.

When participants were asked whether the non-destructible model captured their attention, 50% agreed, 40% disagreed, and 10% strongly disagreed, indicating a generally neutral-to-negative response regarding the attention-grabbing quality of the non-destructible model. By contrast, 90% of participants found the destructible model attention-grabbing, four participants strongly agreed and five agreed, while only one participant disagreed. These results indicate that the destructible model elicited substantially more attention than the non-destructible model in this group sample, suggesting that destructibility may positively influence attentional engagement.

Regarding the clarity of visual details of destruction in the non-destructible model, 50% of participants disagreed, 10% strongly disagreed, and 40% agreed, again reflecting a neutral-to-negative perception of visual clarity. In the destructible model, 80% of participants agreed that the visual details were clear and easy to follow. Of those respondents, agreement was evenly split between strong agreement and agreement. Only 20% found the visual effects in the destructible model unclear or difficult to follow, indicating that the destructible model was generally perceived as easier to follow in visual clarity.

When asked about the realism of the destruction effects in the non-destructible model, 50% of participants strongly disagreed and 40% disagreed that the effects appeared realistic, while only 10% considered them realistic. In contrast, 100% of participants agreed that the destruction in the destructible model appeared realistic. Within this group, 90% strongly agreed and 10% agreed, indicating overall positive leaning. These findings suggest that the destructible model was perceived as more realistic than the non-destructible model, though there remains an upwards trajectory in improving realism.

When directly comparing the two models, participants were asked whether the lack of destruction in the non-destructible model diminished their overall experience: 50% agreed, 20% strongly agreed, and 30% reported that the lack of destruction did not diminish their experience. Conversely, when asked whether the presence of destruction enhanced their overall experience with the destructible model, 70% agreed, 20% strongly agreed, and 10% disagreed. Taken together, these responses indicate that, within this sample, introducing a destructible model had a generally positive impact on participants' overall experience.

We further tested the observed positive-response rate against a theoretical neutral benchmark of 50% to decide wether the pattern in the sample reflects real preference in the population or is just random chance. Following formal hypothesis frameworks were constructed for this matter:

- H_1: The introduction of a destruction model has a positive impact on player engagement towards the system.
- H_2: The simulated destruction effects appeared realistic.

$$H_{01}: \ P(X = 2 \vee 3) \neq 0.5 \qquad H_{02}: \ P(X = 2 \vee 3) \neq 0.5$$
$$H_1: \ P(X = 2 \vee 3) = 0.5 \qquad H_2: \ P(X = 2 \vee 3) = 0.5$$

For our first hypothesis test with a sample of size n = 10 and a significance level $\alpha = 0.05$. The null hypothesis H_{01} states that the destruction model has no effect on perceived player engagement in the population, the alternative states that the destruction model changes engagement.

The right-handed proportion test produced an observed p-value of 0.0057, allowing us to reject the null hypothesis and conclude that there is sufficient evidence to suggest that the introduction has a positive effect on player engagement, significantly exceeding the neural benchmark.

This approach will be the mirrored for our second hypothesis test with the same significance level. The second null hypothesis H_{02} states that the simulated destruction effects appeared to be not realistic, the alternative states that the destruction effects appeared to be realistic.

The right-handed proportion test produced an observed p-value of 0.0029, allowing us to again reject the null hypothesis and conclude that there is sufficient evidence to accept that destruction effect appeared to be realistic, also significantly exceeding the neural benchmark.

6 Conclusion

The destruction model is divided into two distinct stages: geometry preparation and runtime destruction. In the first stage, we focus on constructing fragments of the polytope using Voronoi tessellation. This process starts with calculating a Delaunay triangulation of the 2D projection of the polytope. By employing the duality property of the Delaunay triangulation, we efficiently convert this triangulation into the corresponding Voronoi diagram. Once the Voronoi cells are established, they are extruded into three dimensions, forming the fragments of each object for our runtime destruction.

During the second stage, runtime destruction occurs when a specific impulse threshold is exceeded during collisions involving destructible bodies. When this threshold is reached, the original object is erased, and fragments are spawned at the object's original location. This allows for a seamless transition from a whole body to its shattered fragments.

To effectively showcase the capabilities of our destruction model, we built a simple house model and subjected it to impacts from rigid body boxes. This setup served to demonstrate how the destruction model performs under a practical scenario. To evaluate the effectiveness of our solution, we assessed key performance indicators to measure the execution times of various processes involved in the model. In addition, we conducted a questionnaire involving participants, asking them to judge the realism and their satisfaction with the destruction effects.

The results of our evaluations indicate that the proposed destruction model achieves realistic and satisfactory results. This indicates that real-time destruction is possible using the Vienna Physics Engine.

References

[1] The morgan kaufmann series in interactive 3d technology. In *Real-Time Collision Detection*, C. Ericson, Ed., The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco, 2005, p. iv.

[2] AJEES, S. D. B. A. Real-time procedural modeling of fracture of brittle materials in games, 2009.

[3] Anderson, T., and Anderson, T. *Fracture Mechanics: Fundamentals and Applications*, 3rd ed. CRC Press, 2005.

[4] Arribas, M., Elipe, A., and Palacios, M. Quaternions and the rotation of a rigid body. *Celestial Mechanics and Dynamical Astronomy 96* (2006), 239–251.

[5] Avirup Mandal, P. C., and Chaudhuri, S. Galerkin enhanced graph-based fem for interactive fracture and sculpting applications, 2025.

[6] B.C.Bosch. About the destruction of objects in computer games - a real-time fracture method based on weighted centroidal voronoi diagrams, 2016.

[7] Brown, M. How games do destruction, 2025.

[8] Clothier, M., and Bailey, M. Creating destructible objects using a 3d voronoi subdivison tree, 2015.

[9] Coumans, E. Overview of destruction and dynamics methods, 2011.

[10] Forslund, E. Optimising 3d object destruction tools for improved performance and designer efficiency in video game development, 2023.

[11] Games, E. Chaos destruction, 2026.

[12] Grönberg, A. Real-time mesh destruction system for a video game, 2017.

[13] Grünbaum, B. *Convex Polytopes.* Springer New York, NY, 2003.

[14] Gustafsson, D. Year summary, 2024.

[15] Hlavacs, H. The vienna physics engine (vpe), 2026.

[16] Hlavacs, H. The vienna vulkan engine (vve), 2026.

[17] Institution, S. The father of the video game: The ralph baer prototypes and electronic games - video game history, 2025.

[18] Jeffrey Smith, A. W., and Baraff, D. Fast and controllable simulation of the shattering of brittle objects, 2001.

[19] Kihl, R. Destruction masking in frostbite 2 using volume distance fields, 2010.

[20] Kotsinas, I., and Tholén, V. Voronoi fracturing, 2021.

[21] Ledoux, H. Computing the 3d voronoi diagram robustly: An easy explanation. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)* (2007), pp. 117–129.

[22] Ledoux, H. Tetrahedralisations and 3d voronoi diagrams, 2020.

[23] Leonard Herman, Jer Horwitz, S. K., and Miller, S. *The History of Video Games*. GameSpot, 2002.

[24] L'Heureux, J. The art of destruction in 'rainbow six: Siege', 2016.

[25] Lien, J.-M., and Amato, N. M. Approximate convex decomposition of polygons. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2004), SCG '04, Association for Computing Machinery, p. 17–26.

[26] Lindbergh, B. The destruction (and reconstruction) of destructible environments in video games, 2024.

[27] Lo, S. Parallel delaunay triangulation in three dimensions. *Computer Methods in Applied Mechanics and Engineering 237-240* (2012), 88–106.

[28] Makoto Ohta, Y. K., and Nishita, T. Deformation and fracturing using adaptive shape matching with stiffness adjustment, 2009.

[29] Martin Henk, J. R.-G., and Ziegler, G. M. Basic properties of convex polytopes, 1999.

[30] Matthias Müller, Leonard McMillan, J. D., and Jagnow, R. Real-time simulation of deformation and fracture of stiff materials, 2001.

[31] MORAVCIK, B. Rigid body simulation for academic game engine, 2025.

[32] Müller, M., Chentanez, N., and Kim, T.-Y. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph. 32*, 4 (July 2013).

[33] Oda, O., and Subramaniam, N. Fast dynamic fracture of brittle objects in 3d. In *ACM SIGGRAPH 2006 Research Posters* (New York, NY, USA, 2006), SIGGRAPH '06, Association for Computing Machinery, p. 128–es.

[34] Project, T. C. *CGAL User and Reference Manual*, 6.1.1 ed. CGAL Editorial Board, 2026.

[35] radiangames. Complete technical breakdown of the physics+destruction system in instruments of destruction, 2022.

[36] Rebay, S. Efficient unstructured mesh generation by means of delaunay triangulation and bowyer-watson algorithm. *Journal of Computational Physics 106*, 1 (1993), 125–138.

[37] Richter, J. Destructible environments in control: Lessons in procedural destruction, 2020.

[38] Ronnegren, J. Real time mesh fracturing using 2d voronoi diagrams, 2020.

[39] Rycroft, C. H. Voro++, 2025.

[40] Sekanina, J. An exploration of algorithms for real-time terrain destruction, 2023.

[41] Serway, R. A., and Jewett, J. W. *Physics for Scientists and Engineers*, 6th ed. Brooks/Cole, 2004.

[42] Stegmayr, C. Procedural deformation and destruction in real-time, 2008.

[43] Thomas, R., and Zhang, W. Real-time fracturing in video games, 2022.

[44] van Gestel, J. Procedural destruction of objects for computer games, 2011.

[45] Wikipedia contributors. Bowyer–watson algorithm — Wikipedia, the free encyclopedia, 2025.

[46] Wilder, M. W. An investigation in implementing a c++ voxel game engine with destructible terrain, 2015.

[47] Yan, M., and Wu, D. A new fracture simulation algorithm based on peridynamics for brittle objects. *IEEE Access 11* (2023), 88609–88617.