

Real-Time Fracturing in Video Games

Rachel Thomas¹ and Wenshu Zhang²

¹School of Technologies, Cardiff Metropolitan University

²EUREKA Robotic Centre, School of Technologies, Cardiff Metropolitan University

October 1, 2026

Abstract

Destruction in video games traditionally use pre-fracturing techniques to efficiently swap out game objects at run-time, which requires artists to create multiple models in various states of destruction. Real-time methods require less design time and can produce more realistic results but often come at a cost of performance. If real-time methods can perform at similar levels as tradition pre-fracturing means, this could increase the current standards of realistic destruction in video games. This paper explores and implements real-time fracturing techniques, comparing these to traditional pre-fracturing methods in terms of visual realism and performance. The results of the implementation were then distributed in an online questionnaire, to view how participants perceived the differences of both techniques and whether the visual or performance aspects affected their preferences of one method over the other.

1 Introduction

For games with structures and geometry that are, by design, intended to be destroyed by the player, the destruction system used is very instrumental for successful user experience. Destruction in games can range from small breakable objects containing useful pickups inside, to large structures like buildings where users can destroy different sections to make new entryways. This kind of destruction can be used to create new gameplay strategies, enhancing the user experience; enabling users to dictate their own diverse and dynamic gameplay experiences by directly interacting with the environments they immerse themselves in.

The most common method to create destruction in video games is through pre-fracturing three-dimensional (3D) models, which is the act of pre-defining a ‘broken’ variant of an object and switching the undamaged model with its ‘damaged’ variant when called at run-time [Cou11, VGB11]. Pre-fracturing techniques are predominantly used in video games as they have proven to be very efficient and easy to implement. While pre-fracturing meshes is very efficient, it can have limitations on visual realism. Realism in the video games industry is becoming increasingly sought after, and this is shown through upcoming releases of game engines like Unreal Engine 5, a goal of whose is to achieve photorealism on a similar level to real life [Epi20].

Implementing real-time or procedurally generated destruction can capitalise on the shortcomings of pre-fracturing when it comes to visual realism, though it is often more computationally expensive and affects performance as a result. However, with the increasingly advanced technology being used in current consoles and PCs, it is becoming more achievable to use real-time destruction techniques instead of pre-fracturing. This has the potential to surpass the limits of current destruction techniques and provide future video games a more realistic and immersive player experience.

1.1 Pre-Fracturing and Model Switching

The process of pre-fracturing begins with artists using 3D modelling software such as 3DS Max, Blender, Maya, etc., to create a 3D model then using various techniques to fracture it within the editor before implementing it into a game. After the models are exported, it is within the game that the original ‘undamaged’ 3D model is switched with the pre-fractured version at runtime, which in

combination with the physics engine, can produce aesthetically pleasing destruction behaviours, with realistic rigid body simulations of the broken fragments (Figure 1).

Geometry Preparation

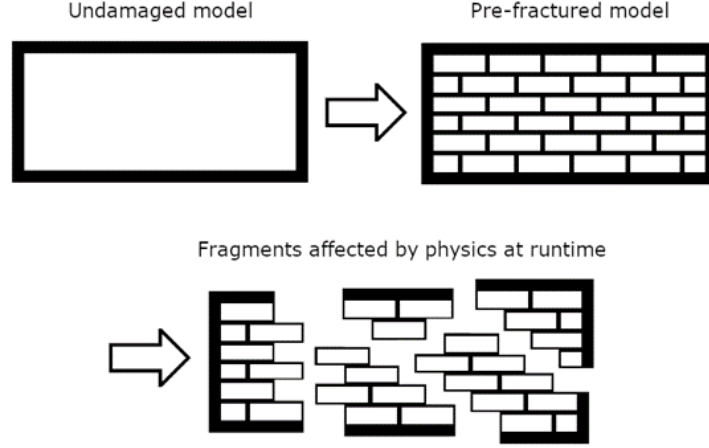


Figure 1: Preparation of 3D geometry

However, a limitation to pre-fracturing is how destruction is represented to the player. When using prefabs or multiple objects of the same kind, they will always fracture or break apart in the same way, which is not a very realistic approach and may become noticeable to the player over time. Additionally, no matter where impact occurs on an object, the whole object will be fractured, and fragments may fall apart synchronously if no constraints are present. This may not be an issue for relatively small objects such as breakables that get destroyed in one hit, but for larger objects that you would expect to have more mass and density, this method may not look very realistic.

Larger in-game structures such as buildings would realistically comprise of several connected components representing different materials that when destroyed by the player, should break apart and collapse in a realistic manner; a building should not completely collapse if the player focuses their fire to a small section of the building, therefore it is also important to consider the placement of constraints to ensure that structures behave realistically when interacted with.

Pre-fracturing is especially evident in games such as the Battlefield series. In Battlefield 3 [DIC11] for instance, players can destroy parts of buildings to create new entryways, or to cause entire buildings to collapse, changing the environment in new and interesting ways.

The Battlefield games visualise this destruction by swapping meshes at runtime. A structure such as a building will comprise of many independent parts (Figure 2), both destructible and non-destructible [Bat19]. When a player destroys part of a building, the geometry of that section of the building will be swapped with its ‘damaged’ version. This mesh swap is disguised using particles like smoke and debris, so the player cannot see the swap occurring.



Figure 2: Battlefield building parts [Bat19]

1.2 Real-time Destruction

Unlike pre-fracturing, real-time destruction involves the fracturing of geometry at run-time. Geometry can break apart in unique ways and can produce different results every time, which is a great way of creating realistic, destructible environments and keeps game-play fresh and dynamic.

Geometry that has not been pre-fractured undergoes calculations at runtime to determine unique fracture patterns, which can take the point of impact into account. While this can achieve more realistic visual results, the time it takes to perform calculations make this a longer process than pre-fracturing. This is a central reason as to why pre-fracturing is favoured over real-time destruction techniques in video games.

An example of real-time destruction can be seen in the game Tom Clancy’s Rainbow Six: Siege [Ubi15], where interior walls and floors can be damaged using explosive or bullets to secure tactical advantages against opponents. Figure 3 demonstrates how players use the game’s core run-time destruction library to full effect by creating a hole in the wall geometry to see into an adjacent room, where opponents can be shot at through.



Figure 3: Rainbow Six: Siege procedural destruction on walls [Rai20]

Players can specifically choose where damage will occur on the surface of the wall while the remainder of the wall remains intact. It would be difficult to predict and display this type of destruction using pre-fracturing methods.

Real-time destruction is not used very often in video games as the current methods of pre-fracturing are deemed adequate for most games, even with the reduced level of realism that occurs as a result. Questionnaire data gathered by [VGB11] found that although game developers considered ‘quality of the results’ as one of the five most important aspects of procedural destruction, realism itself was not deemed as important.

While realism is not considered as important, finding methods of real-time destruction that can perform just as efficiently as pre-fracturing, with the additional benefit of producing higher levels of realism would be an overall improvement for destruction in games.

2 Related Work

After acknowledging the basic workings of both pre-fracturing and real-time destruction techniques, the following section focuses specifically on real-time mesh fracturing techniques that currently exist, and solutions researchers have developed to implement this in real-time. The real-time fracturing research below has been separated into two categories: Real-Time Fracturing Solutions and Pre-defined Fracture Patterns used in Real-Time. The first of these is related to general solutions involved in fracturing meshes with little to no preparation from designers as to how objects will fracture, the second section, pre-defined fracture patterns, involves the real-time application of fracture patterns that have previously been defined by artists.

2.1 Existing Real-Time Fracturing Solutions

A main consideration for real-time mesh fracturing is initially defining how the mesh will fracture; how its fracture patterns are defined and how the new mesh fragments are created.

Generally, meshes can be prepared for fracturing using many different techniques, [Cou11] and [Nie13] describe some of these being: Boolean operations, Voronoi Shattering or slicing, Convex Decomposition, and Tetrahedralization by 3D Delaunay Triangulation. These methods can be used for pre-fracturing, then triggered at runtime and affected by physics, but can also be calculated in real-time.

Boolean operations can be performed in real-time, as seen in games like Red Faction [THQ16], whose Geo-Mod engine allowed players to create tunnels in terrain; altering the initial geometry by subtracting the approximated volume of an ‘empty’ object representing the area of damage caused by the weapon used from terrain geometry [Red].

[PO09] described a system developed for the game Star Wars: The Force Unleashed [Luc08] that could perform mesh fracturing in combination with mesh deformation. Their system is based around a corotational tetrahedral Finite Element Method (FEM), which encapsulates a polygonal mesh into a tetrahedral FEM mesh. Manipulation of the mesh’s matrices when forces are applied to them enables objects to deform and fracture. This technique measures strain applied to the impact area of the mesh; distributing various levels of deformation across its surface. The use of a FEM mesh is more suitable for materials that deform before fracturing, giving materials such as wood and some metals realistic bends before they fracture.

A method of real-time fracturing involving convex decomposition was proposed by [MCK13]. This novel fracturing method revolves around the decomposition of objects into convex shapes using Volumetric Approximate Convex Decomposition (VACD), then using Boolean operations to apply a convex fracture pattern (based on Voronoi) onto meshes. This enables recursive fracturing and partial fracturing around impact points. VACD is useful as it can be applied to non-convex objects to generate a set of convex shapes in its stead. Convexity of objects is important as algorithms and collision systems can run more efficiently on convex shapes opposed to non-convex shapes [Mid88]. As the fracture pattern being applied consists of convex polygons, the fragments produced by the fracturing are also convex in shape.

A technique commonly used to define mesh fragments is through Voronoi shattering/fracturing. Voronoi diagrams can be used to divide 2D planes and 3D meshes into fragments. [Ron20] describes computing 2D Voronoi Diagrams and extruding them into 3D space in real-time, to create dynamic fracture patterns representing glass fracturing. This method is also able to dynamically calculate the fracture pattern based on a point of impact on the mesh surface. Due to calculations being performed in 2D space, this technique is limited to thin planar objects such as flat walls and panes of glass. For meshes that would represent walls or thicker objects, the planar technique would not provide realistic fracturing as there would be no mesh vertices within the volume of the object to provide realistically shaped internal faces. In this case a 3D Voronoi algorithm would be more suitable.

[Grö17] describes how this can be done using 3D Voronoi tessellation on convex objects. Unlike 2D Voronoi fracturing, 3D Voronoi fracturing works by placing random points within the volume of a mesh and using these to create tetrahedral convex fragments. [Täh18] also used a 3D Voronoi technique to create a dynamically changing game environment that could be modified in real-time by the player. Their proposed system utilises Delaunay Triangulation to extract the corresponding Voronoi diagram. As the Voronoi diagram itself needed to be calculated in real-time, an efficient algorithm was required. The most suitable algorithm in this case was the Bowyer-Watson algorithm; [Täh18] stated: “... this algorithm is easy to understand and since it is an incremental algorithm, it has the ability to dynamically change whenever a new point is added without the need to recalculate the entire diagram every time”.

2.2 Pre-Defined Fracture Patterns used in Real-Time

Another method of fracturing a mesh is by applying a pre-defined fracture pattern to geometry at runtime when it is needed. This method gives artists more freedom to create many unique fracture patterns to suit whichever type of material they want. Unique fracture patterns could be designed for wood, concrete, and glass, then be applied to meshes of the specified material.

Re-visiting the game mentioned in Real-Time Destruction, Rainbow Six: Siege uses fracture patterns for its surface procedural destruction. At the Game Developers Conference in 2016, the Technical Lead / Physics Programmer from Ubisoft, provided details on how fracture patterns were used for bullet and explosive impacts on walls and floors [L’H16]. He stated that a cut pattern is generated based on game-play input and material parameters that were set for objects. Figure 4 describes how cuts are

made on a 2D projection of the in-game planar object such as a wall or a floor, after the 2D projection has been clipped via polygon intersection algorithms and re-triangulated, the new output is extruded back into 3D space.

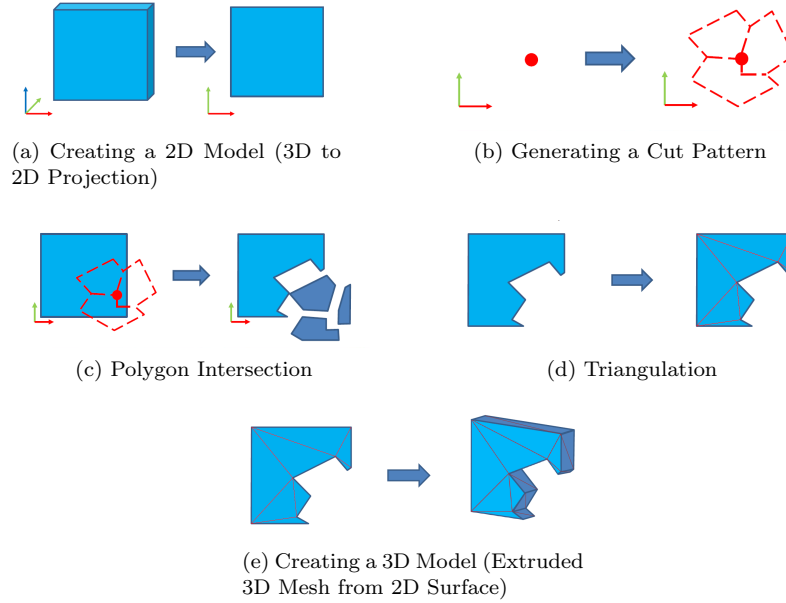


Figure 4: Rainbow Six: Siege Surface Destruction [L'H16]

Many cut patterns used in this game to represent the damage caused by different weapons, so a variety of uniquely shaped fracture patterns are ready to cut wall geometry in real-time.

[VGB11] developed a fracture pattern editor specifically for use in real-time video games. In a similar way as the fracture pattern technique mentioned above, this fracture patterns editor intersects the designed pattern with the mesh then cuts it via Boolean operations. This technique was successful for implementing fracture pattern mesh destruction on basic objects.

The previously mentioned convex decomposition mesh fracturing method by [MCK13] also uses fracture patterns as part of their real-time dynamic fracturing system. The pattern used is based on a Voronoi Diagram, and this pattern is applied to impact areas on the mesh to fracture it. What makes this method more realistic is its recursive nature, as it can re-fracture previously fractured meshes from the first fracturing pass. The method used also maintains the structural integrity of mesh areas that are out of range from the fracturing impact location, allowing partial fracturing.

2.3 Short Summary of Reviewed Works

The various real-time fracturing methods reviewed in each section contain useful techniques that could be used to create a prototype for real-time fracturing. Some methods are more suitable for different video game scenarios than others, and some more applicable to certain types of materials than others. From the techniques discussed above, many of them were able to run in real-time, whereas others were not deemed fast enough. It is crucial that the real-time fracturing techniques used in the proposed experimentations can provide a high level of fracturing realism without creating noticeable performance issues when used in a video game scenario. Those proven to be unable to maintain acceptable performance levels for video games will not be used in the proposed experiments.

[Grö17] pointed out that their 3D Voronoi fracturing implementation was deemed unsuitable for real-time application, as the runtime execution of the 3D Voronoi calculations resulted in a noticeable drop in frame rate. [Täh18] 3D Voronoi destruction system showed successful Voronoi calculations in real time, and even though the type of destruction implemented differs from the topic of this paper, some of the techniques discussed such as Delaunay Triangulation and the Bowyer-Watson algorithms could be useful in implementing an efficient Voronoi fracturing system.

A successful use of Voronoi fracturing was [Ron20]’s 2D to 3D Voronoi fracturing implementation. This was able to fracture an object containing up to 500 Voronoi cells, amounting to 500 meshes, at 60 frames per second (FPS). At this frame rate, this technique has promise for use in video games, even fast-paced games. If a game has a target frame rate of 30 FPS, even more fracturing could occur. To further improve the performance of fracturing calculations, [NTP18] proposed a system to achieve a fast real-time Voronoi implementation deployed on the low-level side - by using GPU. Their work provides a direction for possible extension of the current Voronoi fracturing method to support material-based prorogation and fracturing.

Based on the reviewed literature, the ideal techniques for fracturing thick concrete-like materials would be using 3D Voronoi diagrams as this produces chunk-like mesh fragments. For planar glass-like materials, which will be the subject of testing in this paper, the ideal technique would be based on [Ron20]’s 2D to 3D Voronoi fracturing techniques. This technique works in a similar way to Rainbow Six: Siege’s 2D to 3D projection technique, which is proven efficient enough for video game application, and can also fracture based on impact location: an important element of achieving realistic fracturing. If the Voronoi Diagram can be computed using even faster algorithms such as Delaunay Triangulation with Bowyer-Watson algorithms as mentioned by Täht [Täh18], this could produce realistic fracturing for glass materials very efficiently.

3 Implementation

The implementation was created in Unity3D using C# scripting language. The character controller and framerate counter used were provided by the Unity Standard Assets Package¹. The implementation consists of several experimentations of pre-fracturing and real-time fracturing of various shaped objects including standard 3D cube objects, simple 3D cuboid shapes to represent walls and windows, and on a vase-like object representing a more complex mesh to perform fracturing on.

The implementation is intended to demonstrate pre-fracturing and real-time fracturing behaviours of objects representing different materials: glass, concrete, and wood. This would involve the use of model-switching behaviour for pre-fractured objects whose damaged variants would be created using 3D model editing software, and then creating unique real-time fracturing techniques to apply the appropriate fracturing behaviours to the material type it represents. The final implementation contains pre-fracturing experiments for the materials of concrete and glass, and an experimentation of real-time fracturing – glass material. The game objects representing glass materials are visually compared and performance tested.

Based on findings during the review of literature, mesh fragments for pre-fracturing and real-time fracturing will be defined using Voronoi Diagrams. The sections below describe the main algorithms involved including: Voronoi Diagrams, its dual – Delaunay Triangulation, and the Bowyer-Watson algorithm used to compute Delaunay Triangulations.

3.1 Key Techniques and Algorithms

3.1.1 Voronoi Diagram (VD)

The Voronoi diagram was named after Georgy Voronoi who developed a method of spatial division based on the works of René Descartes in 1644, and later by mathematician [OBS92], as such, it is sometimes known as Dirichlet Tessellation.

VDs enable a 2D plane to be divided into several polygons, which are formed around generating points (one generating point per cell). The various shapes of the cells (Voronoi regions) are determined by the distance from their own generating point in comparison to others – areas within a cell are always closer to its own generating point than to the generating points of neighbouring cells.

The figure below (Figure 5a) by [LZ06] describes the VD as a set of ‘sites’ or ‘points’ in 2D Euclidean space, where each site (P_1, P_2, P_3 , etc..) represents a polygon or the face of a cell.

Each point in a face is equal distance or closer to its own site than to any other. This is highlighted in Figure 5b. All points within the area of face ‘ P_5 ’ are closer/equal distance to its own generator site than any other. The distance calculations from each site are what determines the edges and cell shape.

¹Standard Assets Package (for Unity 2018.4)

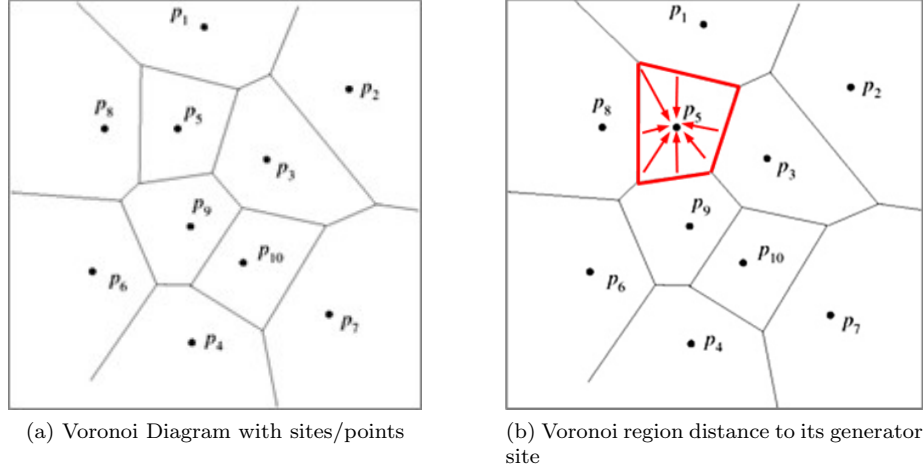


Figure 5: Demonstration of Voronoi Diagram [LZ06]

A benefit of using VDs for object fracturing is that polygons created are always convex in shape and when working with geometry, it is more robust and efficient to run algorithms on convex shapes than on concave shapes [Mid88]. Geometry can be fractured by cutting along the edges of each Voronoi cell.

Though the descriptions of the VD above all reference the use of it on a 2D plane, this can be modified and projected into 3D for applications like visual effects for animations and film, and for video game destruction.

3.1.2 Delaunay Triangulation (DT)

Named after the mathematician Boris Delone, Delaunay Triangulation is considered the dual graph of the Voronoi diagram; A Voronoi diagram can be extracted from a Delaunay Triangulation, and a Delaunay Triangulation can be extracted from a Voronoi Diagram (Figure 6). For example, a site/generator point of a VD corresponds to a vertex of a DT, pairs of edges of DT and VD are orthogonal to each other. The boundary of a DT is the convex hull of a set of points on a plane.

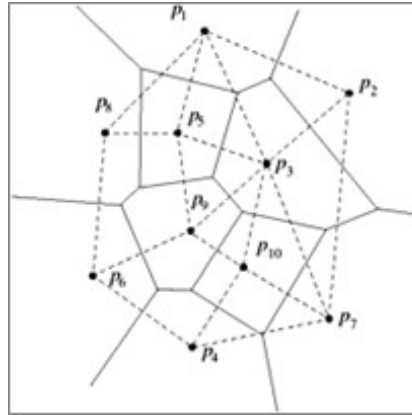


Figure 6: Delaunay Triangulation/Voronoi Diagram dual graph [LZ06]

DT is used to subdivide a plane into triangles, and where other triangulation methods can produce undesirable triangles that are long and thin in shape, Delaunay uses an angle-optimal solution to reduce this called ‘max-min’, which maximises the minimum interior angles of triangles for optimal results, and this is achieved by edge flipping.

[DBCVKO08] discussed how the validity of edges can be determined using Thales’s Theorem, by confirming no other vertices reside within the circumcircle of a specified triangle. Taking the

Voronoi/Delaunay duality example from above, Figure 7a shows what an invalid edge looks like, as the circumcircle of triangle (P_6, P_{10}, P_4) , contains another vertex (P_9), meaning edge $(P_6 - P_{10})$ is invalid, and is sub-optimal triangulation. Figure 7b shows how this edge can be flipped to create a valid edge $(P_9 - P_4)$ with optimal angles, as the circumcircle of triangle P_6, P_9, P_4 , contains no other vertices.

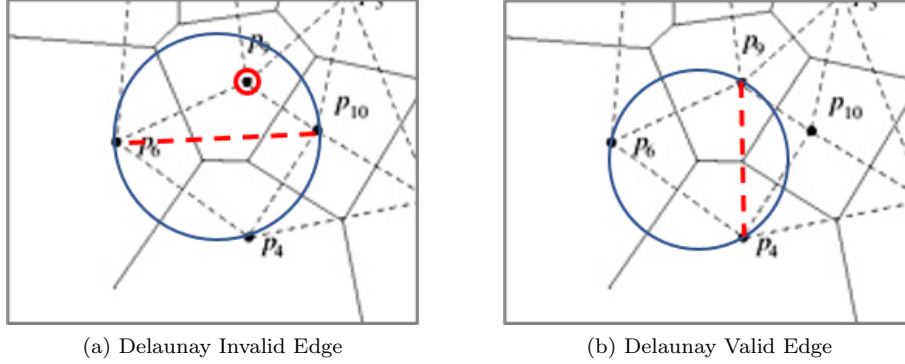


Figure 7: Delaunay Edge.

Using DT is useful for geometry fracturing as it can be used to access the VD, and its angle optimisation solution can prevent undesirable fragments when geometry is cut.

The concepts above can be used in pre-fracturing and real-time implementations, which are discussed in the following sections.

3.1.3 Bowyer-Watson Algorithm

The Bowyer-Watson algorithm is a method of calculating Delaunay Triangulations developed by both [Bow81] and [Wat81]. It is an incremental algorithm that can be used to define new triangles when a new point is added to the overall data structure. This enables the original DT geometry to be updated with newly defined, valid DT triangles in the surrounding area of the new point. This algorithm would be appropriate for real-time fracturing as incremental algorithms are essential for dynamically changing geometry, as unlike other methods, the changes made due to re-triangulation only affects a local area, preventing the need to calculate triangulation across the entire structure [Led07].

3.2 Pre-fracturing Mesh Implementation

Pre-fracturing was a multistep process and involved preparing the initial meshes in 3DS Max then using other readily available Voronoi fracturing tools to separate the mesh into Voronoi fragments. Pieces of software such as Houdini and Blender have built-in Voronoi fracturing tools and were used to prepare the meshes for the pre-fracturing experiment. All meshes were then imported into Unity.

Figure 8 shows how the user can select the number of ‘scatter points’ relating to the number of ‘Voronoi sites’ to include in the fracturing process. Figure 9 shows how this is visualised through colour, and an exploded view to show each individual fragment.

While Houdini was able to create useful visualisations of the Voronoi fragments for meshes, issues when exporting the meshes at the time prompted the decision to use Blender for Voronoi fracturing and exporting the meshes. Like Houdini, in Blender the user can choose how many Voronoi sites to disperse within the volume of the selected mesh, which can be distributed in 3D space along the x , y , and z axes, or sites limited to only two axes, e.g. x and y axes, to create an extruded 2D Voronoi fracture pattern with no internal fragments (Figure 10).

For the concrete wall object, the following 3D Voronoi cell fracture was applied to the mesh inside the editor (Figure 11).

This method created fragments inside the mesh with faces rotated at various angles for a more natural visual effect. The figure below (Figure 12) shows the same cell fracture modifier being used on the vase object to break the mesh into fragments.

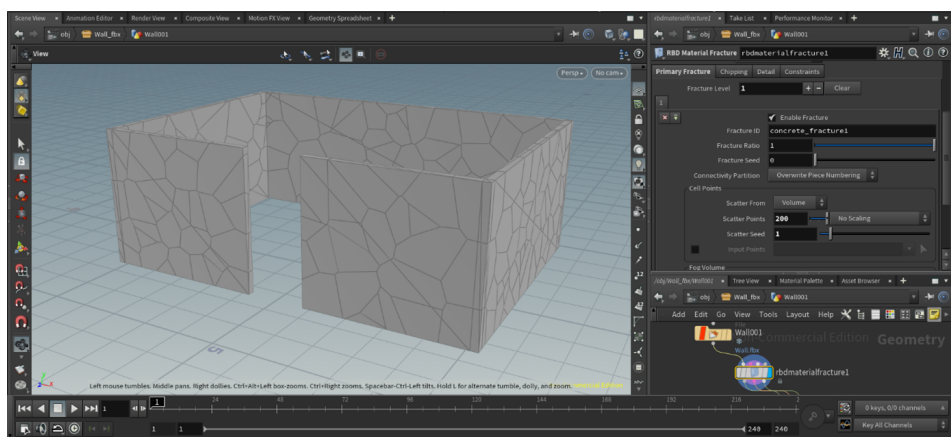


Figure 8: Voronoi fracturing of 'walls' in Houdini

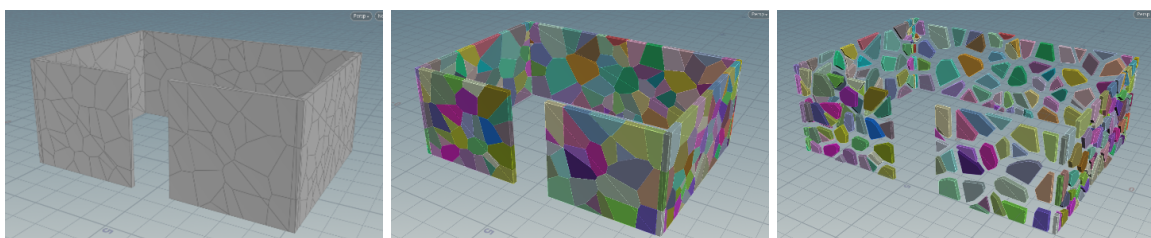


Figure 9: Visualisation of Voronoi fracturing in Houdini

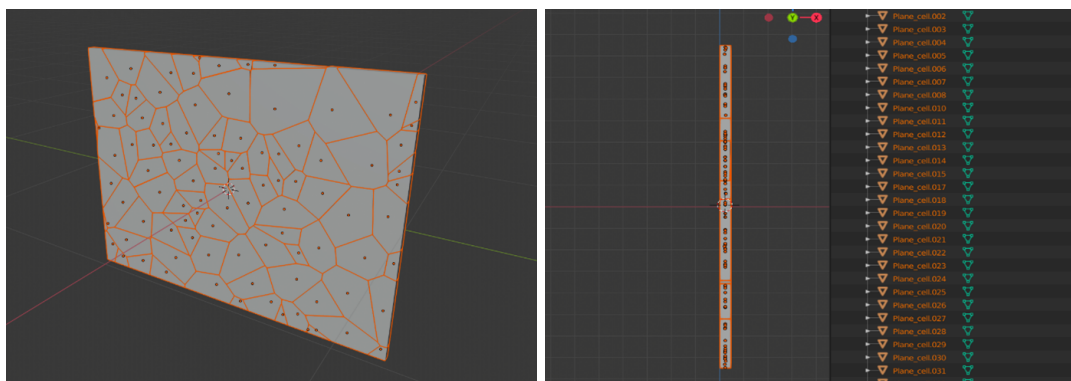


Figure 10: 100 Voronoi sites scattered along two axes of an extruded plane, cutting straight through the mesh

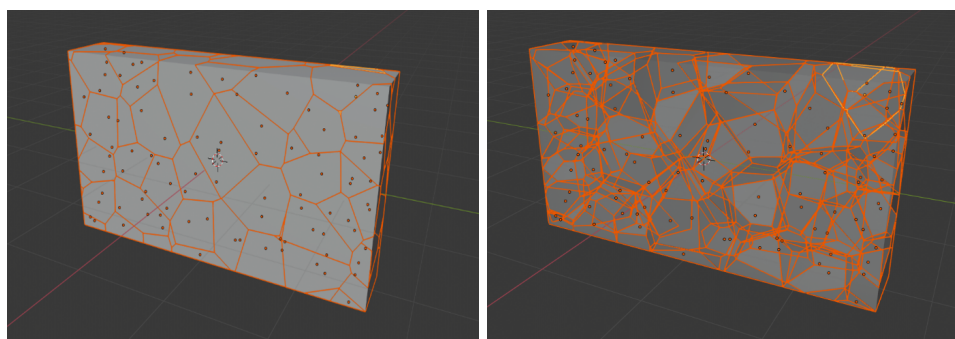


Figure 11: 3D Voronoi cell fracture modifier in Blender containing internal fragments

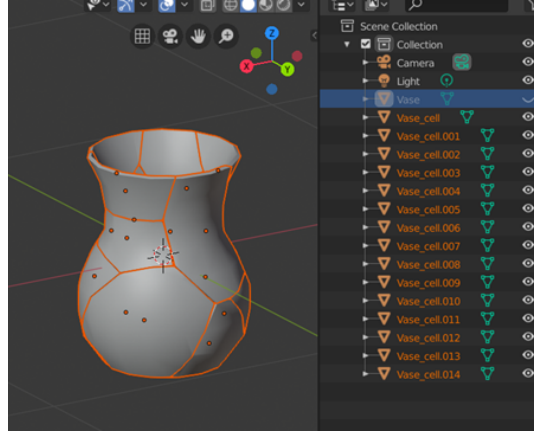


Figure 12: Using Blender's cell fracture tool on a vase-like model

3.2.1 Pre-fracturing Experiments

Pre-fracturing was simple to implement in Unity. Each object intended for pre-fracturing has the following 'DestroyItem' script attached. Once each fragment of the damaged version has a rigid body and convex mesh collider applied, the original models can be switched with this upon collision with a projectile. This script instantiates the damaged variant – a prefab assigned in the Unity inspector, to the position and rotation of the undamaged version (Figure 13).

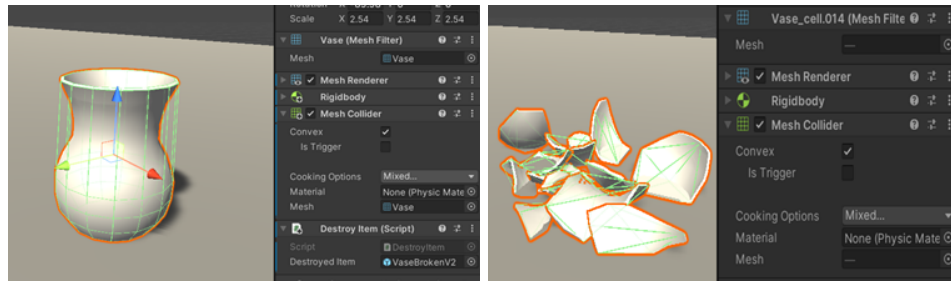


Figure 13: Vase pre-fractured in Unity. Before and after a collision

The effect is almost instantaneous and as such, the switch is disguised very effectively. Although not currently implemented, this action could be hidden completely by activating particle effects during the switch. The same script was applied to the cuboid wall object created in Blender with one-hundred fragments (Figure 14).

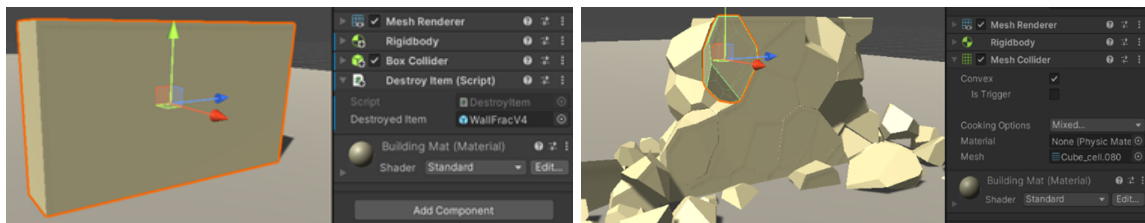


Figure 14: Wall pre-fractured in Unity. Before and after collision

An issue with this method is that each fragment of the damaged variant is essentially 'loose'; fragments all fall due to gravity physics. Adding constraints to keep fragments together using Unity's FixedJoint Component was one of the attempted solutions, however, this created issues with realism where some fragments would unnaturally float in position (Figure 15).

Ultimately, the material used in the final comparison prototype is based on the fracturing of glass materials, the following section shows how this was implemented using pre-fracturing.

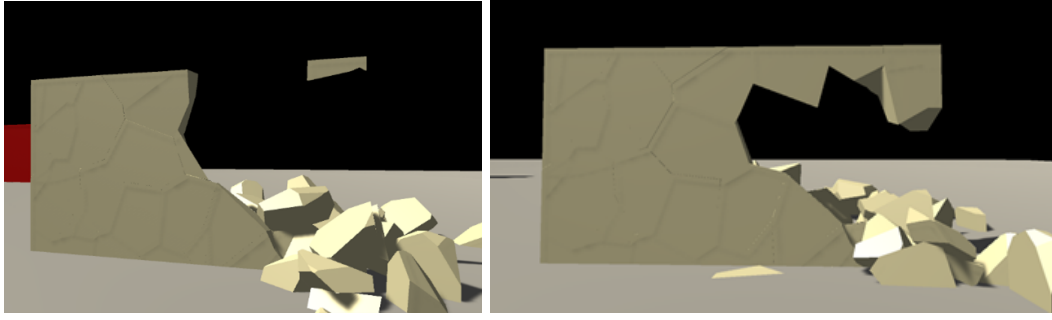


Figure 15: Irregularities using FixedJoint component - floating fragments, unrealistic physics

3.2.2 Pre-fracture Example for Glass Materials

For the main implementation and comparison of pre-fracturing and real-time fracturing, a simple building was created in 3DS Max with multiple gaps to place the ‘glass’ window game objects (Figure 16).

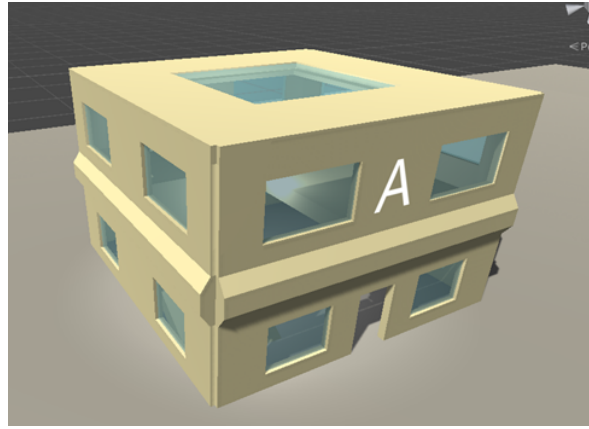


Figure 16: Custom building with glass windows in Unity

Each window in the pre-fracturing example contains the same ‘DestroyItem’ script used in the section above, which replaces the original mesh with a damaged variant. Figure 17 shows each step of the fracturing process when the player fires a physical projectile at the window objects. The first image shows the undamaged window, the middle image displays the instantiated damaged variant, and the product of the final image is created by firing another projectile into the fractured pieces, displacing them.

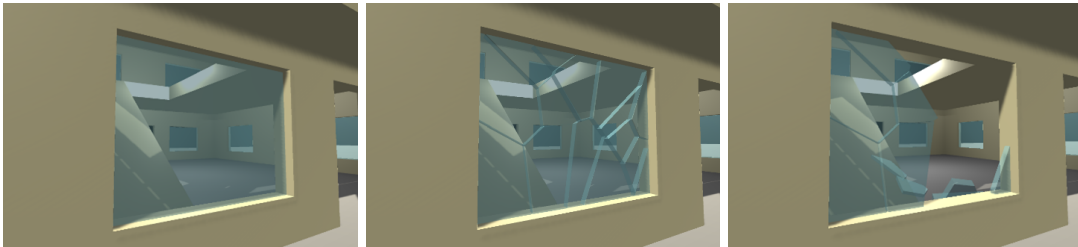


Figure 17: Pre-fractured example of ‘glass’ window. From left to right: Undamaged, damaged, fragments displaced after shooting again

The fracture pattern based on VD remains the same for all windows. This can become noticeable to the player after repeating the action on multiple windows. One note about this implementation as can be seen in the middle image, though it appears that fragments are connected, there are no

constraints between them; the fragments remain together only due to their rigid body components and colliders keeping them within the space of the window.

3.3 Real-time Fracturing Implementation

3.3.1 Real-time Fracturing Example for Glass Materials

The ideal real-time technique for implementing glass fracturing based on the research conducted within the Existing real-time fracturing solutions section, was through calculating 2D Voronoi patterns and extruding them into 3D space. It was also learned that Voronoi diagrams can be calculated using Delaunay triangulation in combination with the Bowyer-Watson algorithm. In this paper, we followed the algorithm proposed in [DBCVKO08], and the Voronoi and Delaunay Triangulation calculators and mesh clippers developed by OskarSigvardsson (2019)².

The implementation uses Delaunay Triangulation and the Bowyer-Watson Algorithm to generate a Voronoi Diagram, the edges of which are cut to create the new mesh fragments. The technique is also recursive as additional fractures can be performed on previously fractured pieces. The number of Voronoi sites can be defined within the source code. It determines the number of fragments created by each fracturing. The number of fragments is set to '15' to match the number of fragments in the pre-fracturing glass implementation.

During the implementation, a mesh will only be able to fracture if its geometrical area is larger than the stated minimum break area variable set in the Unity Inspector or within this script. This offers more control to which fragments can be re-fractured based on their size. This is very useful as it prevents the generation of too many meshes or fragments that are very small in size, which would affect overall performance.

One of the ways this implementation creates realistic fractures is through generating the Voronoi diagram at the point of where impact occurs. This creates a higher concentration of Voronoi sites around the impact point, resulting in smaller fragments towards the centre of impact and larger fragments towards the outer edges of the mesh, this can be seen in Figure 18, where impact has occurred on the left-hand side of the mesh.

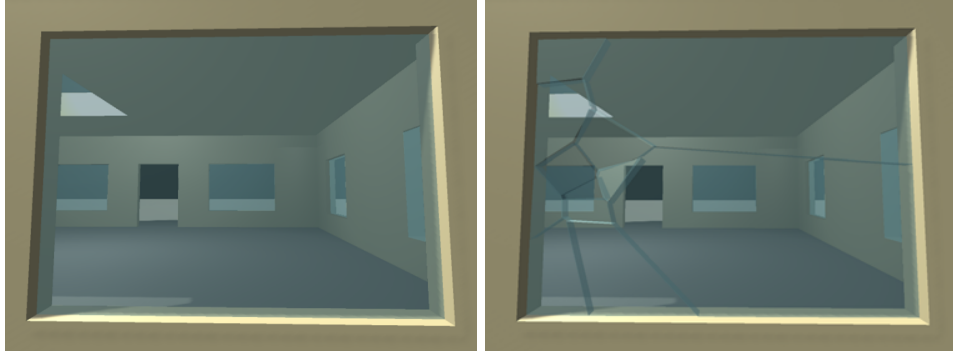


Figure 18: Real-time fracture applied to impact area on the left side of the window

In a similar way to pre-fracturing, the calculated fragments are instantiated in place of the original 'parent' mesh (which is destroyed), taking the local position and rotation into account then adding a 'Rigidbody' component to the fragments so they can be simulated and interacted with in the scene. The implementation verifies Delaunay Triangulation by checking points within the circumcircle of triangles, and by checking for valid/legal triangulation edges and flipping them if they are not valid. After that, the Bowyer-Watson algorithm is then run to return newly created triangles caused by the addition of another point in the diagram. The Voronoi Diagram is then calculated based on the new triangulation of the mesh and then clipped to create the new fragments.

All the algorithms work together to create very visually realistic fracturing for glass materials, particularly when reducing the minimum break area of the mesh to allow for smaller fragments.

²<https://github.com/OskarSigvardsson/unity-delaunay>

4 Results

In video games, one of the most important statistics to measure performance is linked to frame rate or FPS. The FPS value equates to the number of frames that are rendered to the screen in the span of a second. The higher the FPS, the smoother games appear to run. If this number is low, the game will noticeably stagger and can become unplayable.

The target frame rate for most video games is usually set at 30 frames per second, for games that require low latency this target is increased to 60 FPS [MS18]. There is a framerate counter visible during runtime that shows the current FPS, but to get a broader, more in-depth view of the performance of fracturing, the ‘Profiler’ tool within the Unity editor was used. The Unity Profiler tool can measure CPU usage, GPU rendering usage, memory allocations, physics performance, and other resources on the device while the prototype is running.

4.1 Implementation Performance testing

To test performance of both the pre-fracturing and real-time fracturing implementations, a development build of the prototype was created and run alongside the Unity profiler. The experiment is tested on both a basic laptop and a latest high-spec PC (see Table 1 for details) to investigate the in-game performance of various devices. A simple test was conducted, consisting of shooting two projectiles at one window of the building model described in last section, and analysing how it affects components of the system and the overall frame rate.

Component	Low Specification	High Specification
OS	Windows 10	Windows 10
CPU	Intel(R) Core(TM) i5-8250U CPU @1.60GHz, 1800 Mhz, 4 Core(s), 8 Logical Processor(s)	Intel(R) Core(TM) i5-10400F CPU @2.90Ghz, 2904 Mhz, 6 Core(s), 12 Logical Processor(s)
GPU	(Integrated) Intel(R) UHD Graphics 620	NVIDIA GeForce RTX 3060ti 8GB GDDR6
RAM	8GB, DDR4, 2400MHz	32GB, DDR4, 3200MHz
HDD	1TB 5400rpm	1TB M.2SSD read 2400Mb/s, write 1950Mb/s

Table 1: Device specifications used for implementation

4.1.1 Pre-fracturing Implementation Performance on Low-Spec Laptop

The following image (Figure 19) shows the profiler results at the time both projectiles were fired at one of the window objects:

In Figure 19, the impacts of both projectiles fired at the window correlate to the distinct response seen in the Physics module, and the increased Rendering activity in the profiler.

The frame rate can be seen to oscillate between 30 FPS and 60 FPS (Figure 20). The action of instantiating the fractured object into the scene and simulating the rigidbodies did not appear to cause any notable deviations to the average frame rate, or cause frames to drop lower than 30 FPS.

After removing the filters of other elements that were executed during fracturing, leaving only the graphics rendering, scripts, and physics elements (Figure 21) visible, the speed each element performed at can be seen. Rendering the graphics took 3.28 milliseconds (ms), the execution of scripts took 1.06ms, and physics calculations took 6.54ms.

4.1.2 Pre-fracturing Implementation Performance on High-Spec PC

As with the high-end device’s pre-fracturing profiling image, the moments of impact of the projectile to the window can be seen by the two distinct rises in the physics module of the profiler (see Figure 22).

The performance in terms of framerate remained over 60 FPS, at approximately 75 FPS throughout the experiment with no drops in framerate (Figure 23). Unlike what was present in the low-spec experiment, the framerate was consistent throughout.

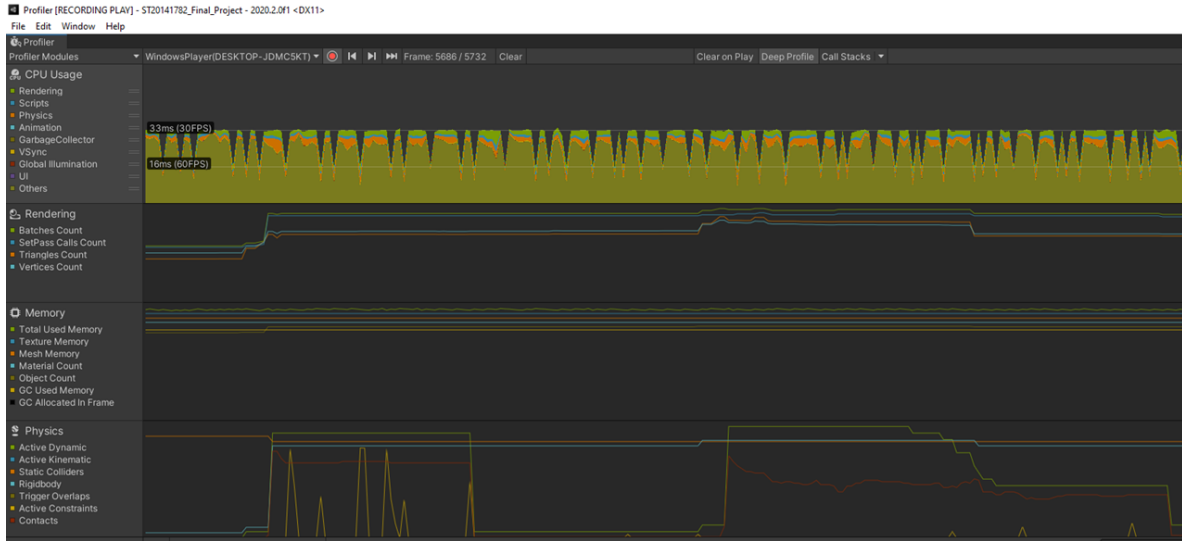


Figure 19: Profiler results for pre-fracturing performance test

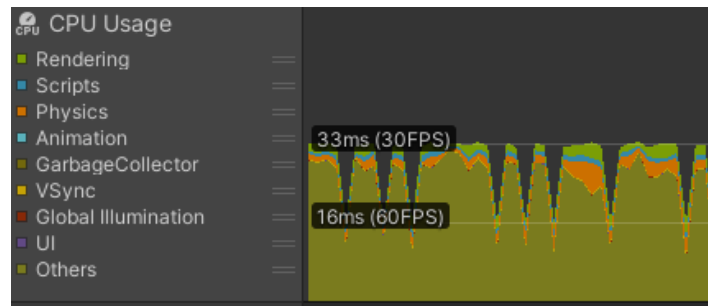


Figure 20: Frame rate for pre-fracturing test

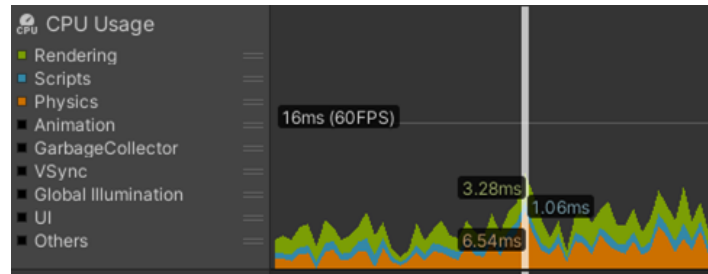


Figure 21: CPU usage for **Rendering**, **Scripts**, and **Physics** in the pre-fracturing test

Isolating the rendering, scripts, and physics components revealed that at the time of impact, rendering took 0.40ms, scripts took 0.18ms to execute, and the physics took 0.49ms to calculate (Figure 24).

4.1.3 Real-time Implementation Performance on Low-spec Laptop

In the same format as the pre-fracturing test, the image below shows the overall performance of firing two projectiles at a single window in the real-time fracturing implementation:

While the average overall FPS can be seen to vary between 60 and 30 FPS, there are three notable spikes in activity when both projectiles were fired at the window – two spikes during the first collision, one spike at the second. This caused frame rate to temporarily drop below 30 FPS to approximately 22-23 FPS. Again, removing elements other than the Rendering, Scripts, and Physics, helps visualise how each of these performed (Figure 26).

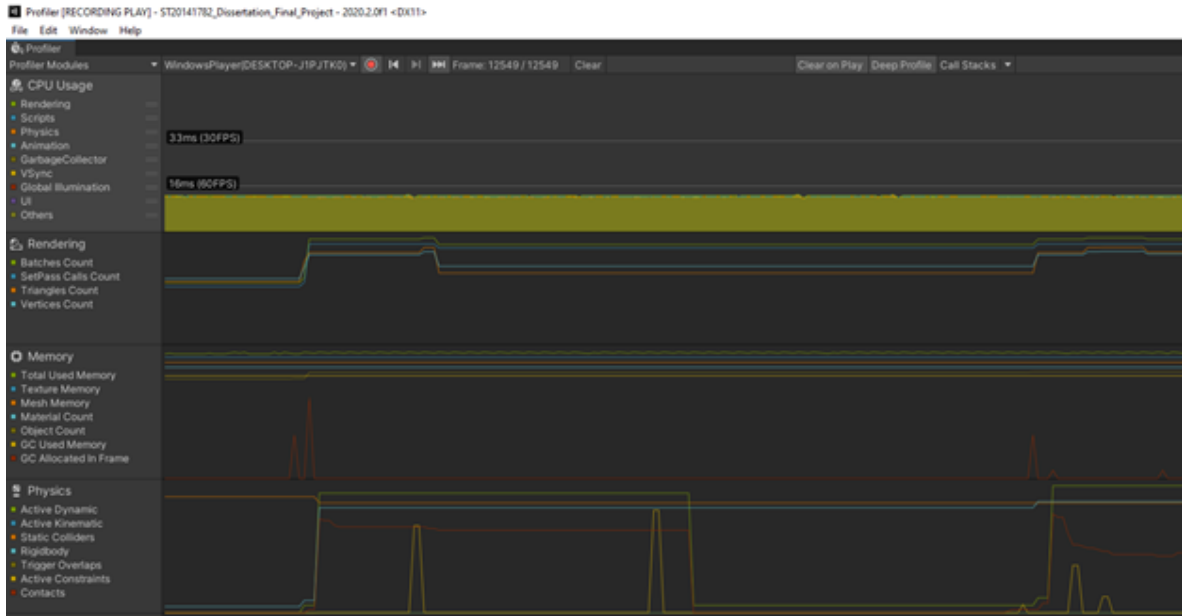


Figure 22: Profiler results for pre-fracturing performance test

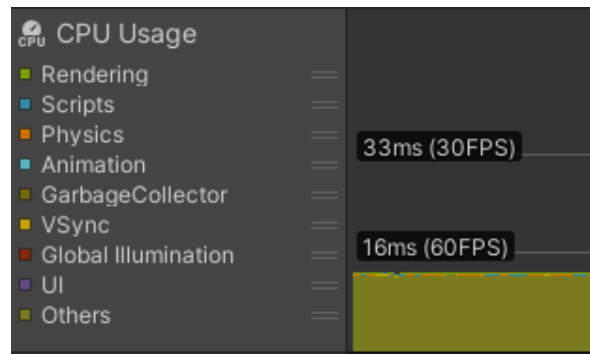


Figure 23: Frame rate for pre-fracturing test

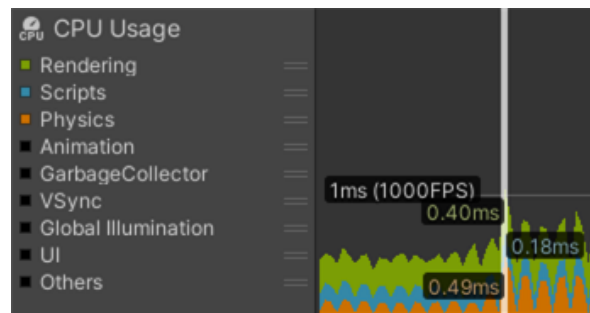


Figure 24: CPU usage for **Rendering**, **Scripts**, and **Physics** in the pre-fracturing test

Figure 27 shows how much time each element took in the frame when initial fracturing occurred: Graphics rendering took 3.08ms, script executions took a 18.59ms, and physics amounted to 6.05ms.

4.1.4 Real-time Implementation Performance on High-spec PC

The real-time fracturing performance on the higher-spec PC achieved a similar FPS baseline as with the pre-fracturing test, sustaining a stable 75 FPS with minor deviations. The only notable difference, which can be seen in the graph below (Figure 28), is the spike during the first collision signifying a



Figure 25: Profiler results for real-time fracturing performance test

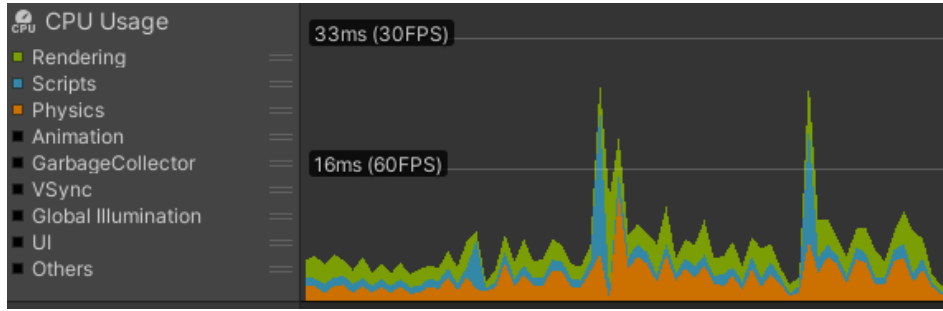


Figure 26: CPU usage for **Rendering**, **Scripts**, and **Physics** in the real-time fracturing test

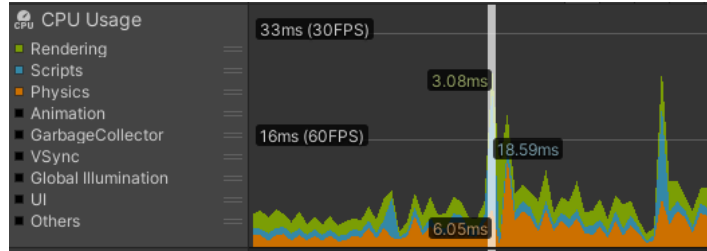


Figure 27: CPU usage for **Rendering**, **Scripts**, and **Physics** at the time of the initial fracture (real-time fracturing test)

drop in framerate to approximately 65 FPS.

Analysing the spike by isolating the rendering, scripts, and physics modules shows that 0.37ms was spend on rendering, 5.00ms on script execution, and 0.50ms on physics (see Figure 29).

4.1.5 Discussion of Performance results

For the experiment on the low-spec laptop, the performance tests show that both the pre-fracturing and real-time fracturing implementations performed very similarly with slight differences in performance. The pre-fracturing test was able to perform within the ideal frame rate range of 60 to 30 FPS throughout, whereas the real-time fracturing test had frame rates that temporarily dropped below 30 FPS to 22-23 FPS whenever fracturing occurred. This is not ideal as it would be a noticeable to the player in a game. Depending on how often the player fractures the glass, this drop in FPS will continue



Figure 28: CPU usage for **Rendering**, **Scripts**, and **Physics** in the real-time fracturing test

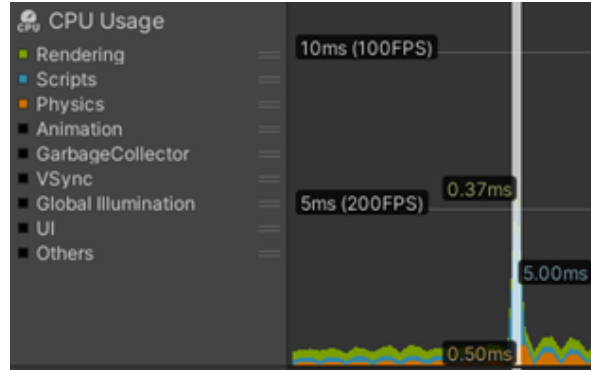


Figure 29: CPU usage for **Rendering**, **Scripts**, and **Physics** at the time of the initial fracture (real-time fracturing test)

to occur.

Both experiments performed on a higher-spec PC were able to perform consistently over 60 FPS. There were no performance issues with the pre-fracturing test, and even with the spike in performance loss coming from the real-time fracturing test, the overall framerate stayed above 60FPS, without dropping below this. As anticipated, comparing the script execution times of both the pre-fracturing and real-time fracturing techniques still shows a significant difference between script execution times with real-time's 5.00ms to pre-fracturing's 0.18ms. However, as previously mentioned, the more powerful hardware was able to handle this without any significant drops in FPS.

Performing the same experiment on both a low-spec and high-spec PC provided a useful insight into the limitations of real-time fracturing regarding hardware requirements. Where the low-end PC suffered performance loss to a standard that would be considered unsuitable for use in video games with its sub-30 FPS framerate, the high-end PC was able to manage the additional demands of the real-time fracturing implementation and maintained a framerate above 60 FPS. Table 2 shows performance results comparison between the low-spec PC and high-spec PC for both pre-fracturing and real-time fracturing experiments.

The question remains of whether this framerate can be maintained on the more powerful hardware when more than a single instance of real-time fracturing takes place. In a dense environment where multiple real-time fracturing events take place simultaneously, the accumulation of script executions may prove to impact performance in a significant way.

4.2 Questionnaire

A questionnaire, created using Qualtrics³, was distributed to groups of users familiar with technology and video games. The primary target group being university students studying technological subjects such as robotics, computer science, and computer games design and development. The questionnaire was distributed to the target demographic, students and lecturers alike, and completed anonymously.

³Qualtrics(2021). The Leading Research & Experience Software.

Performance results on a low-spec PC		
Profiler Module	Pre-fracturing	Real-time Fracturing
Rendering	3.28ms	3.08ms
Scripts	1.06ms	18.59ms
Physics	6.54ms	6.05ms
Performance results on a high-spec PC		
Profiler Module	Pre-fracturing	Real-time Fracturing
Rendering	0.40ms	0.37ms
Scripts	0.18ms	5.00ms
Physics	0.49ms	0.50ms

Table 2: comparing performance results of pre-fracturing and real-time fracturing on both a high-spec PC & a low-spec PC

The questions from the questionnaire and responses to these questions can be found in the supporting materials (Appendix A and Appendix B).

The broad structure of questions involved having participants watch short videos of the fracturing techniques on glass materials described in *Implementation*, then answering questions about them. The decision to use pre-recorded videos to demonstrate the fracturing techniques was made due to difficulties that would arise by distributing build files online in large numbers, and due to current restrictions involving physical access for participants to test the implementation on public devices. Using videos however inhibits the collection of valuable feedback on the performance of the implementation on computers with different specifications.

The type of fracturing techniques: pre-fracturing and real-time fracturing, were not mentioned in the questionnaire to received unbiased feedback; they are differentiated as technique ‘A’ and ‘B’ respectively. Then questions were asked about the visual realism, accuracy to the material, and the perceived performance of each technique. Finishing with two questions about its practicality in video games based on their opinions.

In total, 20 participants completed the questionnaire. 19 out of 20 stated they played video games ‘Very Often’ or ‘Often’, with 1 stating they did not play often. This indicated that all participants were familiar with video games.

4.2.1 Pre-fracturing Questions

The second question was about the pre-fracturing implementation video. 16/20 of respondents thought the visual realism of destruction was ‘Somewhat realistic’, so results were neither ‘very realistic’ or unrealistic but at a level of realism between these. 4 out of the 7 additional comments left for this question stated they noticed the windows fracturing with the same pattern, confirming it as one of the main shortcomings for pre-fracturing and a reason for its reduced realism. As for the fracturing representing glass, again 16/20 of respondents thought it represented glass ‘Somewhat realistic’(ally), however three chose ‘Not very realistic’. Some of the reasons for this, again, was the noticeable fracture pattern. Another commented on the ‘thickness’ of the glass – a valid observation, as this was not considered fully at the time of implementation.

The perceived performance of pre-fracturing was an even split between ‘Moderately fast’ and ‘Very fast’. As anticipated, this confirms it is an efficient technique to use with no significant drops in frame rate. Only one comment suggested they saw some ‘slowdown’ when the large ‘ceiling glass’ was broken.

4.2.2 Real-time Fracturing Questions

The questions for the real-time fracturing demonstration followed the same format as the pre-fracturing technique.

In terms of visual realism, 12/20 of respondents thought the real-time fracturing demonstration was ‘very realistic’, with 7/20 saying it was ‘somewhat realistic’. Only 1 respondent stated it was not at all realistic. Respondents noticed that the fracture patterns on the glass appeared to be different each time it was broken, and that the glass could be broken repeatedly. A couple of respondents also noticed that fracturing occurred at the point of contact with the window.

In terms material accuracy, it was a close divide between ‘very accurate’ and ‘somewhat accurate’ with 9 votes for the former and 10 votes for the latter, with 1 respondent voting it was not at all accurate. Respondents again pointed out the randomness of shattering, its ability to fracture again, and how the glass breaking at the point of impact portrayed the glass accurately.

For performance, the top two choices were split by only one vote: 9 respondents thought the technique performed ‘very fast’, and 8 chose ‘moderately fast’. However, 3 respondents noticed some performance issues and selected ‘slow’. Even though most respondents perceived the technique performing quickly overall, based on the comments left, most noticed small drops in frame rate when actions such as the large ‘skylight’ glass broke and when increasingly more glass was broken.

4.2.3 Final Closing Question Results

To conclude the questionnaire, participants were asked which technique they preferred overall, taking the visual realism and performance aspects into account. This question was asked to see if users would value performance less for a higher level of realism or would choose improved performance over realism. The question was an open question, as such, it was necessary to collate each respondent’s technique of choice from their answers.

In total 16/20 of respondents chose technique ‘B’, 3/20 chose technique A, and one respondent did not state a preference as they believed both were relatively similar and the use of each technique would depend on the intended scenario.

The main reasons given by participants in their preferred choice of Technique B (real-time fracturing), over Technique A (pre-fracturing), was for its realistic and ‘detailed’ fracturing – ability to break again into smaller fragments, and the randomness of fracture patterns created from the origin of impact. Several respondents acknowledged a small performance drop yet still chose this technique. Those that chose Technique A stated the faster performance of this technique over technique B was a primary reason for their choice, and that the visual realism of glass fracturing was acceptable.

The final question asked of the suitability of using either technique, specifically for video games. Generally, responses were favourable for both techniques to be used in games, with many emphasising how technique B would offer more ‘visually pleasing’ and realistic results.

Some important aspects concerning performance in video games were also mentioned, such as how technique B may ‘tax system resources’; and that ‘heavy lag’ that may be caused by techniques. One response pointed out that Technique A would ‘scale better as the game expanded’ and would be ‘easier to maintain’. Another mentioned that technique A is able represent glass breaking even though it is not as realistic as technique B.

4.2.4 Discussion of Questionnaire Results

From the bar charts of the visual realism of both techniques (supporting materials - Appendix C), technique B is evidently the most popular choice, this was due to its ability to fracture recursively, fracture with random fracture patterns, and fracturing from an impact location.

For material accuracy (supporting materials - Appendix D), again Technique B contained the highest number of votes for the best accuracy option in comparison to technique A, which only had a single vote for this. However, a large portion of respondents chose ‘somewhat accurate’ which is still an acceptable level of accuracy.

Technique A had the best perceived performance results (supporting materials - Appendix E), with respondents only choosing ‘Very Fast’ and ‘Moderately fast’. Although most respondents also chose ‘Very Fast’ and ‘Moderately fast’ for Technique B, ‘slow’ was also chosen by a few. Most noticed some drops in frame rate – correlating with results from the performance testing, which is not ideal in video games. There was a large response in favour of Technique B (real-time) when respondents were asked which technique they preferred. This data is visualised in a pie chart (supporting materials - Appendix F). This demonstrated that users would like to see this level of realism in video games more than traditional pre-fracturing techniques.

The overall interpretation of the questionnaire confirms that higher realism comes at a cost of performance, but in this case the severity of the performance issues, although noticeable, was very low. With optimisations this could be reduced further to provide high levels of realistic glass fracturing and improved performance.

5 Conclusion and Future Work

Many real-time fracturing techniques were studied from the research conducted in this paper. In comparison with other techniques, generating Voronoi fractures using Delaunay Triangulation with the Bowyer-Watson algorithm has the advantages of calculating fracture patterns for game objects representing glass materials and creating fractures based on impact locations for additional realism.

One of the key contributions of this work is the investigation it carries out to find out the computational performance of real-time fracturing technique applied in video games. An implementation comparing pre-fracturing and real-time fracturing of glass materials was successfully created and performance tested. The experiments found that both pre-fracturing and real-time fracturing techniques can hold a stable framerate around 75 FPS on a high-spec PC. Although no significant FPS drops are observed from the real-time fracturing technique, there is a noticeable difference between script execution times when comparing it to the pre-fracturing technique. On a low-spec PC, the tests showed that the pre-fracturing technique ran between 60 and 30 FPS, with the real-time fracturing implementation suffering some frame drops, dropping to approximately 22-23 FPS at the time of fracture.

Another contribution of this paper is to observe the viewed visual differences between used techniques, by gathering feedback via an online questionnaire about the visual realism and perceived performance of the techniques used in the comparison. Based on qualitative analysis, most participants noticed the performance issues of the real-time fracturing implementation yet still selected it as their preference over that of the more efficient pre-fracturing method. These pointed out the advantages and practicability of its superior accuracy and realism of the material. This is an indicator that, with optimisations, this real-time fracturing technique has potential to improve the standards of realism in video games for destruction of simple planar objects like windows.

The main limitation of real-time fracturing technique is the run-time performance. While it is relatively fast, the temporary FPS drops are still slightly noticeable and need optimisations. This could be improved by limiting how many times an object can be fractured and how many fragments can be produced from a mesh in total. Fragments created could then be deleted if they are small in size, or if they are inactive after a certain amount of time, to reduce strain on the physics engine.

In the future, we intend to implement the fracturing of other types of materials such as concrete and wood. Additional real-time fracturing techniques could be implemented for these and used alongside the glass fracturing technique to create even more realistic, destructible environments for video games, which would be a significant improvement for realism in video games.

References

- [Bat19] Battlefield Wiki. Destruction. <https://battlefield.fandom.com/wiki/Destruction>, 2019. Accessed: 2021-07-01.
- [Bow81] Adrian Bowyer. Computing dirichlet tessellations. *The computer journal*, 24(2):162–166, 1981.
- [Cou11] Erwin Coumans. ACM Siggraph 2011 overview of destruction and dynamic methods, 2011.
- [DBCVKO08] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Delaunay Triangulations*, pages 191–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [DIC11] DICE. Battlefield 3. [video Game] Playstation 3. Electronic Arts, dice. <https://www.ea.com/games/battlefield/battlefield-3>, 2011.
- [Epi20] Epic Games. A first look at unreal engine 5. <https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5>, 2020. [online; accessed 01/07/2021].
- [Grö17] Anton Grönberg. Real-time mesh destruction system for a video game. <https://www.diva-portal.org/smash/get/diva2:1115491/FULLTEXT01.pdf>, 2017.
- [Led07] Hugo Ledoux. Computing the 3d voronoi diagram robustly: An easy explanation. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 117–129. IEEE, 2007.

- [L'H16] Julien L'Heureux. The Art of Destruction in "Rainbow Six: Siege". <https://gdcvault.com/play/1023003/The-Art-of-Destruction-in>, 2016.
- [Luc08] LucasArts. Star Wars: The Force Unleashed. <https://www.starwars.com/games-apps/star-wars-the-force-unleashed>, 2008.
- [LZ06] Elmar Langetepe and Gabriel Zachmann. *Geometric data structures for computer graphics*. AK Peters/CRC Press, 2006.
- [MCK13] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Transactions on Graphics (TOG)*, 32(4):1–10, 2013.
- [Mid88] Alan E Middleditch. The representation and manipulation of convex polygons. In *Theoretical Foundations of Computer Graphics and CAD*, pages 211–252. Springer, 1988.
- [MS18] Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2018.
- [Nie13] Jens-Kristian Nielsen. Modelling objects for simulation of breakables. 2013.
- [NTP18] Yousif Ali Hassan Najim, Georgios Triantafyllidis, and George Palamas. dynamic fracturing of 3d models for real time computer graphics. In *2018-3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON)*, pages 1–4. IEEE, 2018.
- [OBS92] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial tessellations: concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., 1992.
- [PO09] Eric G Parker and James F O'Brien. Real-time deformation and fracture in a game environment. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 165–175, 2009.
- [Rai20] Rainbow Six Wiki. Bullet Penetration. https://rainbowsix.fandom.com/wiki/Bullet_Penetration, 2020.
- [Red] Red Faction Wiki. Geo-Mod. <https://redfaction.fandom.com/wiki/Geo-Mod>. [online; accessed 07/12/2020].
- [Ron20] Jerry Ronnegren. Real time mesh fracturing using 2d voronoi diagrams, 2020.
- [Täh18] Marko Täht. Real-time cave destruction using 3d voronoi. 2018.
- [THQ16] THQ Nordic. Red Faction. <https://thqnordic.com/article/red-faction-back-and-literally-still-breaking-ground-playstation4>, 2016.
- [Ubi15] Ubisoft. Tom Clancy's Rainbow Six: Siege. <https://www.ubisoft.com/en-gb/game/rainbow-six/siege>, 2015.
- [VGB11] J Van Gestel and R Bidarra. Procedural modelling of destructible materials. In *Proceedings of V Ibero-American Symposium on Computer Graphics, Faro, Portugal*, pages 1–3, 2011.
- [Wat81] David F Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The computer journal*, 24(2):167–172, 1981.