A Comparison of Fracture Techniques
For Use in Video Games

Author:
Jack Davidson,
H00103121

Supervisor:
Dr. Stefano Padilla

Second Reader:
Dr. Patricia A Vargas

A thesis submitted in fulfilment of the requirements
for the degree of MSc.

in the

School of Mathematical and Computer Sciences

August 2018
Declaration of Authorship

I, Jack Davidson, declare that this thesis titled, ’A Comparison of Fracture Techniques For Use in Video Games’ and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Jack Davidson

Date: 16/08/18
**Abstract**

The use of destruction in video games can provide a number of benefits depending on the experience. For some games, it can be a little touch that adds to the immersion, for others its a great way to drive adaptive gameplay.

To achieve this effect of fracturing materials and structures a number of methods can be utilised. Some use physically driven models based on fracture mechanics, such as the finite element method. Other physically based methods mimic reality through the use of a mass-spring system to simulate the fracture without the computational expense of more realistic methods. While some look to achieve the visual appearance of the effect through geometry-based models that ideally generate an effect that is good enough but at a significantly cheaper cost.

The aim of this project is to compare a number of methods for modelling fractures in computer graphics, for use in real-time applications and to provide developers with an enhanced insight into the differences in the performance and quality of the destruction achieved with each.

This was completed through implementing a standard pre-fractured model and a mass-spring model within Unity. Their performance was measured at runtime in the same environment, on the same models. To attempt to provide a more accurate comparison of their runtime performance than what is currently available in literature and to evaluate if there is a perceived improvement through the use of more expensive methods both in terms of computation and for the level of difficulty to implement.

The results show that the pre-fractured method was faster than the mass-spring method and that there does exist a perceived improvement in the quality of the destruction obtained through the more computationally expensive mass-spring method.
Acknowledgements

I would like to thank my supervisor Dr Stefano Padilla, whose advice was crucial in keeping me on track and getting the most out of this project. I’d like to thank Matt Stroud, for putting up with being told every little detail, sometimes with context, while he was busy trying to complete his own project. Finally, I’d like to thank Sarah Williamson, my editor in chief once again, for finding the time to meticulously proof-read this thesis, and correct my many mistakes. I promise I’ll write less for you to check over... next time.
# Contents

Declaration of Authorship i  
Abstract ii  
Acknowledgements iii  
Contents iv  
List of Figures vi  
List of Tables vii  

1 Introduction 1  
1.1 Background and Justification ........................................ 1  
1.2 Aim and Objectives .................................................. 2  
1.3 Scope ......................................................................... 3  
1.4 Outline ......................................................................... 3  

2 Literature Review 4  
2.1 Background ................................................................. 4  
2.2 Geometry-Based Models .................................................. 6  
2.2.1 Pre-Fractured ............................................................ 6  
2.2.2 Dynamic ................................................................. 8  
2.3 Physically-Based Models .................................................. 10  
2.3.1 Mass-Spring Models ................................................... 12  
2.3.1.1 Time integration in mass-spring methods .................. 15  
2.3.2 Finite Element Method ............................................... 19  
2.4 Evaluation ..................................................................... 22  
2.5 Conclusion .................................................................... 23  

3 Methodology 26  
3.1 Environment and Tools ................................................... 26  
3.1.1 Hardware .................................................................. 26  
3.1.2 Tools ....................................................................... 27  
3.1.3 Generating Tetrahedral Meshes .................................... 27  
3.1.4 Loading Tetrahedral Meshes ....................................... 27  

iv
Contents

3.2 Methods ......................................................... 29
  3.2.1 Pre-Fractured ........................................... 29
  3.2.2 Mass-Spring ............................................. 32
3.3 Evaluation ...................................................... 36
  3.3.1 Performance .............................................. 36
  3.3.2 Quality ................................................... 38
  3.3.3 Survey .................................................... 38
  3.3.4 Legal and Ethical Issues ................................. 39
3.4 Finite Element Method .......................................... 40
  3.4.1 The Mathematical Method ................................. 40
  3.4.2 Implementation in Unity ................................. 46
3.5 Conclusion ....................................................... 49

4 Results and Discussion .......................... 50
  4.1 Performance .................................................. 50
  4.2 Survey ....................................................... 54
  4.3 Conclusion ................................................... 57

5 Conclusion ...................................................... 59
  5.1 Future work .................................................. 60

A Finite Element Method Supplemental Algorithms ......... 62
  A.1 Conjugate Gradient Method ................................. 62
  A.2 Inverse of a Matrix .......................................... 64
  A.3 Polar Decomposition Algorithm ............................. 64
  A.4 QR Decomposition and Algorithm ........................... 65
  A.5 Compressed Sparse Row List ............................... 65
  A.6 Coordinate List ............................................. 66

Bibliography ....................................................... 67
List of Figures

2.1 Example destruction in Rainbow Six: Siege .......................... 5
2.2 Example of pre-fractured destruction in Rainbow Six: Siege ........ 7
2.3 Before and after a spiderweb pattern was applied to a window when shot in Rainbow Six: Siege ............................................. 9

3.1 Example of implemented pre-fractured destruction, illustrating a simple shape collapsing ................................................ 29
3.2 UML diagram for the Prefractured class. ................................. 30
3.3 Example of implemented mass-spring destruction, showing the frames from a collision event .......................................... 33
3.4 UML diagram for the MassSpring class. ................................. 33

4.1 Comparison of mean time to render each frame .......................... 51
4.2 Comparison of mean instantaneous framerate ............................. 51
4.3 Standard deviations of framerate ............................................ 53
4.4 Pre-fractured: The destruction of this object looks realistic .......... 54
4.5 Mass-Spring: The destruction of this object looks realistic .......... 55
4.6 Pre-fractured: Why does the destruction look realistic? ............... 56
4.7 Mass-Spring: Why does the destruction look realistic? ............... 56
List of Tables

3.1 Summary of variables for performance experiment .................................. 37
# List of Algorithms

1. Read .vol file .................................................. 28
2. Initialise Prefractured object .................................. 30
3. Generate fragments for pre-fractured object .................. 31
4. Generate mass-spring object ..................................... 34
5. Solving Equation 3.5 for use in conjugate gradient analysis . 48
6. Finite Element Method ........................................... 48
7. Conjugate gradient method ....................................... 63
8. Inverse of a $3 \times 3$ matrix ..................................... 64
9. Polar decomposition .............................................. 64
10. QR decomposition ............................................... 65
11. QR Algorithm ................................................... 65
Chapter 1

Introduction

1.1 Background and Justification

The use of destruction in video games can provide a number of benefits depending on the experience. For some games, it can be a little touch that adds to the immersion through a response to the players hitting a crate or smashing a window. For others, it can be a core component, driving adaptive gameplay in both single-player and multiplayer experiences, giving players a way to carve their own path through the game world or allowing players to take advantage of their evolving environment to get the upper hand. As time goes on an increasing number of games are going to look to take advantage of the concept for immersion or as a core part of their system dynamics.

There are many techniques that exist to create this effect, all to varying degrees of physical accuracy and efficiency. It is critical to review the techniques that are available in order to provide a greater insight into how these techniques perform and how the quality of their results compare to their alternatives. As video-games are highly complex applications that have strict expectations for performance, they have to be able to respond to the player’s input in real-time and keep a steady framerate, unlike stress analysis tools in engineering that are set up and left to perform their simulations over extended periods of time. This means as developers continue to strive to make their experiences more realistic and immersive, they have to be aware of the performance and try to find some balance between the performance and the realism of their scene.
These methods for simulating fracture tend to fall into the categories geometric-based and physically-based [Muguercia et al., 2014]. Geometric-based aim to ideally generate an effect that is good enough, without the same level of computational expense. While physically-based methods look to simulate the fracture mechanics by calculating the forces acting within materials to determine if they will create cracks and if these cracks will propagate. A third category does exist, example-based, it uses computer vision techniques to extract information from an image of a crack to duplicate it on the model.

While surveys have been done to compare and contrast proposed methods [Muguercia et al., 2014, Nealen et al., 2006]. Only [Muguercia et al., 2014] have looked to compare their performance. This was achieved using the data from the original papers to approximate their performance based on the hardware used and the complexity of the models, rather than implement these methods and test their performance in the same environment.

1.2 Aim and Objectives

The aim of this project is to compare a number of methods for modelling fractures in computer graphics, for use in real-time applications and to provide developers with an enhanced insight into the differences in the performance and quality of the destruction achieved with each.

This is to be achieved by completing the following objectives:

- Compile and review the literature.
- Implement a pre-fractured method.
- Implement a mass-spring method.
- Explore the implementation of the finite element method.
- Run performance tests on the pre-fractured and mass-spring methods.
- Obtain semi-quantitative information on the quality of destruction from each tested method.
- Perform significance testing to find if there is a difference in performance from one method to the other.
• Perform significance testing to find if there is a perceived improvement through the use of more expensive methods in terms of computation.

1.3 Scope

While this field expands many different approaches to simulating fracture, this project is to be limited to the comparison of the pre-fractured and mass-spring methods. While the implementation and qualities of the physically-based finite element method will be explored, this method requires a large amount of optimisation before it is capable of real-time use and therefore, will be omitted from the comparison as its implementation is impractical in the timeframe of the project. While methods such as the example-based as discussed in [Muguercia et al., 2014] are omitted, due to their use being more applicable for applying cracks to surfaces to add detail to models, than for performing fracture on an entire object.

1.4 Outline

Chapter 2: Literature Review details some of the key literature in the field, analysing the merits to each method and its applicability to real-time applications, as well as providing the reader with a foundation of the inner workings of each method. Chapter 3: Methodology, goes into detail on how the pre-fractured and mass-spring methods were implemented, tested and evaluated, as well as going into detail on how the finite element method could be implemented and provides an in-depth discussion on each of the steps to complete the simulation. Chapter 4: Results and Discussion, presents the results of the evaluation techniques and discusses what they say about the two methods. Finally, Chapter 5 concludes the thesis with a discussion of any potential future work on the topic.
Chapter 2

Literature Review

In this chapter, a brief overview of the topic is presented and critically evaluated. From this, the methods to be implemented can be chosen.

2.1 Background

Many games strive to build realistic worlds to immerse their players in the experience they are trying to craft. As technology advances, games are able to do more to reach the goal of creating virtual reality. Destruction in games is one part of this. It can be a small touch that adds to the scene allowing the world to react to the player’s actions, such as hitting a ceramic vase and it responding by shattering, shooting a window to see the cracks appear in a web-like pattern, or chipping away at an enemy’s cover. Little details like this allow the player to feel part of a living game world. This, however, is not the only reason for using destruction in video games, some actively use destruction as a procedure to allow players to achieve their objective. Series like EA’s Battlefield use destruction as a means to have the play space evolve as a round progresses, allowing players to remove their enemies cover to gain the upper hand. Ubisoft’s Rainbow Six: Siege lets players blow holes in walls to open them up to allow new ways of moving around the map or to provide new sight lines to spot an enemy from as shown in Figure 2.1. While Volition built their entire game, Red Faction Guerilla, around the procedure, providing their players with a myriad of tools to take down buildings and bridges on top of their adversaries. As well as crafting timed challenges that task the player with taking
down structures with a limited toolset, pushing the player to think about what are the weak points of the structure that they can use to reduce it to rubble efficiently. As time goes on, more and more games are going to look to take advantage of the concept to add to their experience, for immersion or as a core part of their system dynamics. There are many techniques that exist to create this effect, all to varying degrees of physical accuracy and efficiency. Therefore, it is critical to review the techniques that are available in order to provide a greater insight into how these techniques perform, and how the quality of their results compare to their alternatives.

A few reviews of this topic have been conducted over the years [Muguercia et al., 2014, Nealen et al., 2006]. While [Nealen et al., 2006] looked to provide an overview of the significant methods in the field, [Muguercia et al., 2014] went one further to present a survey of the topic with the goal of classifying each method and providing a comparison of the performance of each, something that had not been completed previously. They describe the three main categories to modelling fracture mechanics as being physically based, geometry-based and example-based. Physically-based models simulate the forces within the model to calculate when and where cracks will occur within the material, as well as calculating when the cracks will propagate. Geometry-based aim to mimic the appearance of fractures through a pattern-based approach and are more concerned with computational speed than physical accuracy. Finally, example-based approaches use computer vision techniques to mimic cracks of real-world objects in the models.

While [Muguercia et al., 2014] do provide a strong summary of each method, they do not attempt to implement the methods they discuss. Leading to the comparisons on their performance they provide being only approximations, taking into account the results of their respective papers, the complexity of the models used and the hardware the tests were conducted on. This project looks to further this research by taking example
methods from the first two categories, physical and geometry based, implementing and testing them on the same models, to create a more precise comparison of these techniques. For the geometric approach, implementing the pre-fractured method, and for the physically-based, the mass-spring method. As for the finite element method, while the implementation and testing of this method are out of the scope of this project, the process of implementing it will be explored, as the use of this engineering-based method may prove to be beneficial in the future.

2.2 Geometry-Based Models

Geometric models can be further broken down into the sub-categories, pre-fractured and dynamic [Muguerzia et al., 2014, Muller et al., 2013].

2.2.1 Pre-Fractured

Pre-fractured models are one of the more popular methods in the games industry due to their fast computation at runtime, and for the level of control, the developers have over the look of the destruction [Müller et al., 2013, O’Brien, 2000]. When an object is hit by a force, the magnitude is checked against a critical value for the object. If the force is greater than the critical value, the object is simply replaced by its fragments. Figure 2.2 presents an example of this method in action, as can be seen when the object is hit, its fragments are placed in the scene and the physics is allowed to take over to pull them onto the floor. This method provides direct control over the details of the destruction, from how many pieces the object breaks down to and how accurately the fragments represent the initial object.

There are a few sacrifices made for the runtime performance of this simple approach. The destructible object will break into the same pieces each time it is hit with a significant force. This can be countered by creating multiple variations of the fragments and choosing at random which set to use when the fracture is initiated. However, this introduces its own problems such as an increase in production time spent by 3D artists creating assets, and a larger amount of memory has to be reserved at runtime, to store each of these variations.
A disassociation exists between the impact location that causes fracture and the locations of the cracks, as there is no way to predict the impact location of the object, as it is to be interactive with the user. This can lead to a jarring effect if, for example, a player was to shoot a window, they would expect the fracture to spread from the impact location not elsewhere.

This method can be optimised further for performance and memory at runtime by breaking the object into many of the same fragments that are instanced, instead of generating a high number of unique fragments. While the use of a few basic fractured pieces or the use of Voronoi diagrams to generate the fragments can speed up the authoring time [Müller et al., 2013, Ning and Li, 2010, Raghavachary, 2002] and provide suitable results depending on the kind of fracture that is sought after. [Yan et al., 2010] shows an efficient approach to generating a Voronoi diagram from a tetrahedral mesh and outputting
an even 3D mesh that could perhaps be used for crumbling stone. However, their results show that this method is not fast enough to be used during runtime.

[Ning and Li, 2010] attempts to increase the realism of a Voronoi based pre-fracture in games and simulations by using information of the impact, specifically the location and relative velocity, to determine the normal distribution of Voronoi nodes across the object, with a higher density of nodes around the centre of the impact area. This results in a fracture with smaller fragments closer to the impact location with the pieces getting progressively larger as the distance from impact increases. This, however, relies on prior knowledge of the impact during the creation of the asset, which as stated previously cannot be predicted when dealing with an interactive medium such as video games, therefore, limiting this in a video games context to in-game cinematics that are rendered in real-time but follow a set script. [Ning and Li, 2010] suggest using the conservation of momentum to calculate the velocities of the fragments after fracture by splitting the difference in energy across all the newly created pieces, as another step to increase the realism of the method and prevent the object from just looking like it fell apart. This all results in a method that is very flexible and allows for developers to have control over the tradeoff between runtime performance, memory usage, and authoring time to decide how they want an object to fracture.

2.2.2 Dynamic

Procedural methods look to fix some of the issues of pre-fractured models at a reduced cost on performance than the physically based methods. Methods such as [Müller et al., 2013] do so through the use of fracture patterns. Instead of creating the fragmented objected beforehand a fracture pattern is created. This pattern can be lined up with the impact location, before breaking the volume of the object into fragments based on where the boundaries of the fracture pattern intersect with the mesh. Utilised properly this can solve some of the issues with pre-fractured models like the misalignment of cracks to impact location or the amount of time needed to create fragmented objects beforehand, as it requires only for the artists to generate fracture patterns that can be used multiple times for different types of objects. A spiderweb pattern could be applied to any glass materials, take for example Figure 2.3, while something less even could be used for something less predictable. A major benefit of the use of 3D patterns for the procedural
creation of fragments is that this method can continuously fragment pieces into smaller chunks. This model allows the same patterns to be applied again but scaled down to the appropriate size to create smaller pieces from the fragment repeatedly. Whereas for pre-fractured models to break down a fragment into further fragments requires the artists to create more assets. [Müller et al., 2013] breaks down the 3D model into convex pieces to allow for fast computation of where the fracture patterns intersect with the volume. This paired with the ability to break the fracture patterns down into near and far cells allows for local fracture of the object by welding the fragments designated as far, back together, instead of breaking the entire object, like with the standard pre-fractured model. This ratio of close to far on the fracture pattern could be adapted to be based on the magnitude of the force of the impact, so that the harder an object is hit the more material is broken apart or by selecting the pattern based on the impact force to break the material down into smaller or larger pieces. This concept of keeping pieces farther from the impact together after fracture could be applied to pre-fractured models such as Voronoi based methods like [Ning and Li, 2010] using the force of the impact to determine a radius at which pieces out with the radius will be welded, while the pieces within are allowed to split apart. To allow for the generation of smaller pieces closer to the impact location like [Ning and Li, 2010] attempted to replicate, a higher resolution mesh could be paired with gradually welding more pieces together the farther from the impact location. This would, however, require a larger amount of memory to keep track of a mesh with enough granularity to do this at any point in the object.

A problem that persists in this topic and not just for these models, is that breaking down an object into fragments means there are more objects, vertices, and surfaces to be computed and rendered each frame. This leads to a reduction in the performance and makes it more difficult to hit the required frame rate that is expected for smooth
Chapter 2. Literature Review

gameplay. While most methods are naturally limited to the number of fragments they can be broken down to based on the initial tetrahedral mesh they are given, the method presented by [Müller et al., 2013] is not. If the total amount of fragments is not kept in check through having them disappear or keeping the number of times a fragment can be broken down set, it could quickly cause problems with keeping a steady performance.

2.3 Physically-Based Models

Geometric models are limited in the way they fracture, due to the simplifications they make. Typically they assume fractures occur at the impact location unless specifically hardcoded to occur elsewhere in a pre-fractured model. When fracture does occur it is instant across the material. For collision type events this can be effective however for situations such as pulling the material in tension it does not accurately simulate the typical behaviour.

Take, for example, a cylindrical element of uniform thickness along its length with equal and opposite forces applied to each end pulling the cylinder in tension. The force on each end is gradually raised until it reaches the critical value when the material fractures. Using a basic pre-fractured model, the object shatters into a predetermined number of pieces. With the procedural models discussed previously, taking the points where the forces are applied as the impact points, the object would shatter around where the force is applied. While these are possibilities, it is highly dependent on the material of the object to get these results. Something that neither of these models takes into account. Every object, no matter the material, fractures in an instant, and require alternative patterns or fragments to be created to try to mimic how different materials break.

In reality for the above example, the forces applied to the cylinder would create stress within the material, that in turn would cause the material to deform before breaking. There are two critical values for the stress a material can experience; the yield, and tensile strengths. As the stress applied increases up to the yield strength, the material is said to be deforming elastically, meaning any deformation that occurs up to the yield strength can be recovered from. In essence, once the stress is removed, the cylinder returns to its original state. If the stress surpasses the yield strength of the material, it is said to be deforming plastically, meaning that any increase in deformation past this
point is permanent. Increase the stress further, and the material eventually reaches its
tensile strength, at which point it fails and fracture occurs.

The values of the yield and tensile strengths vary drastically from material to material
and their values determine along with a number of other variables how a material behaves
under applied loads. Brittle materials such as glasses and ceramics have a very small
gap between the yield and tensile strength, so there is little to no plastic deformation
of the material before fracture leading to sharp edges on fracture, like the results found
through geometric models. Take for example the composite, carbon fibre reinforced
polymer (CFRP) this is listed as having a yield strength between $550 - 1050 \text{ MPa}$ and
the exact same for the tensile strength [Ashby et al., 2010], this is a material known
for being extremely strong but when it fails, it fails in an instant as there is no plastic
deformation past its yield strength.

For more ductile materials, that have a larger gap between the yield and tensile strength,
the process is more complex, as the stress is brought above the yield strength, the
cylindrical element would begin to neck around the middle, in essence, the cross-sectional
area would start to shrink. As the stress is force per unit area this reduction in the area
causes an increase in the stress local to the necking. However as the material deforms
plastically it is said to work harden, absorbing the stress making the material local to
the plastic deformation more resistant to further deformation. Eventually, the tensile
strength is reached and the element fails. Unlike with brittle materials where the fracture
is sudden, once the yield strength is passed, a ductile material deforms plastically before
fracture meaning the break isn’t clean, resulting in the material tearing rather than
shattering. An example of a ductile material is aluminium alloys, that are listed at a
range of $30 - 500 \text{ MPa}$ for the yield strength and $58 - 550 \text{ MPa}$ for the tensile strength
[Ashby et al., 2010], meaning that this material does not fail straight after surpassing
its yield strength, however it will show signs of damage from the high applied load and
in many cases have to be removed from use.

By looking at how a typical element fails when in tension shows how geometric models do
not accurately portray fracture. They do not take into account any elastic deformation
before yielding and cannot portray the plastic deformation that occurs after. This
limits their use to materials like ceramics and glasses that are generally brittle with a
high stiffness meaning they are resistant to any deformation.
Physically based models used in computer graphics that originate from engineering literature [Imagire et al., 2009, O’Brien and Hodgins, 1999], look to simulate the behaviour of materials up to fracture but sacrifice some numerical accuracy to get faster computation times as they aim to replicate the visual appearance of fracture rather than simulate when the fracture occurs to help mitigate risk. There are two subsets of which this paper will be focussing on, mass-spring and finite element method.

Both methods discretise the object that is to be simulated down into smaller more manageable chunks, before using laws of motion and fracture mechanics of engineering literature to determine if the fracture is to occur. However, they do so in very different ways.

### 2.3.1 Mass-Spring Models

Mass-spring models break the objects down into a set of nodes connected to their neighbours through springs and dashpots [Mugueria et al., 2014]. These springs provide a restoring force calculated using Hooke’s law:

$$ f = k\delta l $$

(2.1)

where $f$ is the resulting force from the spring of stiffness $k$ that undergoes a change in length $\delta l$. The dashpots provide a dampening force to the system, to resist the motion of the spring through:

$$ f = cv $$

(2.2)

where the force is the result of the dashpots decay constant $c$ multiplied by the relative speed $v$. These forces are typically summed with any external forces such as gravity or those from collisions to be used with the equations of motion, to move each node in the system. This use of springs to connect nodes allows for this method to simulate the elastic behaviour of materials that geometric models could not. Once the force is removed from the element the springs will bring the object back to its original orientation with the help of the dampening to settle the system. To model fracture once the length of a spring has passed a critical point, the spring is said to be broken and is removed
from further calculations to prevent it providing further restoring forces allowing for nodes to split from the rest.

An example of the flexibility of the mass-spring model is how it handles the imperfections that exist in all materials in the form of cracks. These cracks cause concentrations of stress around their tips, which lead to fracture earlier than even the yield strength of the material. This is a big problem in the study of fracture mechanics in engineering but is not considered in geometric models. [Norton et al., 1991] showed that mass-spring models can take this sort of behaviour into account through the random selection of springs in the system and lower their threshold to break. This could be taken a step further by changing the threshold for failure of springs to be different in compression than it is in tension, allowing for proper simulation of heterogeneous materials like concrete, instead of the simplifications made by [Norton et al., 1991] that only allow the fracture to occur when in tension. Another possibility that could be utilised is a different stiffness, dampening constant and the threshold for failure based on the direction the spring is acting in. Some materials are known as anisotropic, meaning they have different material properties dependent on the direction applied. For example, composite materials such a carbon fibre, are strong in tension, however, can withstand very little shear force. This modification could be implemented through a vector for the material at which it is strongest, then calculating the difference between the angle of this vector and the direction of the spring, to get a lowered critical value.

The detail of the resulting fracture of mass-spring methods depends on the elements used within the object. [Norton et al., 1991] used nodes in groups of eight to form cubes, with the nodes connected to some of their neighbours, and cubes attached to the next through their shared nodes. These cubes could be smoothed on surfaces to provide a rounded effect. However this resulted in jaggy edges when fracture did occur as noted by [O’Brien, 2000] this could be countered by increasing the resolution of the grid by using a larger number of smaller pieces, but this would have an adverse effect on performance.

[Imagire et al., 2009] implemented the Extended Distinct Element Method (EDEM) effectively a mass-spring method, for use in computer graphics, that originally was developed in engineering literature by [Meguro and Hakuno, 1989]. EDEM places spherical elements throughout the object and instead of grouping elements into cubes, each element is used as the site for a 3D Voronoi diagram, to generate the mesh for each element.
from the Voronoi faces. Merging the meshes as long as the springs that connect them are not broken, providing a more natural look to fragments when fracture does occur as compared to the jagged cubes of [Norton et al., 1991].

[Bo et al., 2011] apply pre-fracture patterns to the objects, to create the fragments that make up the object as a whole. At runtime, these are held together with a mass-spring system. Leading to an improvement over pre-fractured models, instead of the whole object breaking at once it is now based on a physical model. This however still suffers from the fragments being based on the fracture pattern used. [Bo et al., 2011] use the example of applying a scatter pattern emanating from the impact point on glass, however, again this relies on prior knowledge of the impact location which is not possible in an interactive medium.

While the quality of the simulated destruction can be highly dependent on the resolution of the mesh, too low and the fragments created are large and blocky, too fine and the memory needed for the object can become excessive. [Imagire et al., 2009] suggests a method of generating destruction on three levels, coarse, debris, and dust. Allowing for a higher quality destruction without requiring to increase the resolution of the mesh too far. A coarse breakdown of the object is done using EDEM. Once this is calculated, the energy used to break each element is used to find the maximum size of the debris, if smaller than the EDEM element then the element is broken down further into pieces within the maximum. Finally, the amount of dust created is based on the fracture energy and simulated using fluid dynamics. This process allows for more detail in the destruction, however, the simulation of all three levels proved too slow to be viable for use in real-time applications.

While this method might not use stress to determine failure as in fracture mechanics, the change in length with respect to the original length is known as the engineering strain. Engineering strain changes linearly with the stress applied to the material during elastic deformation, meaning this method of determining when the fracture occurs can be linked back to the stress that the material is experiencing. However, this method does not take into account plastic deformation, limiting it to the simulation of brittle materials like the geometric models. [Terzopoulos and Fleischer, 1988] looked at modelling inelastic, viscous and fracture by using springs, dashpots, and slip units. Springs, as discussed, provides a restoring force, while dashpots provide dampening to the system. Slip units
are components that when an object is given a deformation above the yield point of the material, a fraction of this deformation is provided to the slip unit to prevent the object from returning to its initial state thus taking into account an approximation of plastic deformation. [Terzopoulos and Fleischer, 1988] suggests taking combinations of these three units to create the differential equations that determine the behaviour of the material, to then be discretised through finite difference or finite element methods. Typical mass-spring methods already use the finite difference differential equations of springs and dashpots, the small addition of slip unit could allow for the simulation of ductile materials. A developer could specify what combination of spring, dashpot and slip units are used in a specific object to get an accurate representation of the material’s deformation and fracture.

2.3.1.1 Time integration in mass-spring methods

A problem with the mass-spring model is that every fragment in the object is connected to its neighbouring fragments through a set of springs. With each spring there is displacement that creates a restoring force that effects the equations of motion, Equations 2.3 and 2.4 of the masses connected.

\[ M \frac{dv}{dt} = \sum f \quad (2.3) \]

\[ \frac{dx}{dt} = v \quad (2.4) \]

Thus creating a large number of equations that are required to be solved every frame. How they are solved, is a massive component of this methods viability, and is a popular topic of research in itself [Liu et al., 2013]. [Norton et al., 1991] and [Imagire et al., 2009] both used the explicit Euler method to solve each of these equations, Calculating the sum of the forces from the springs using Equation 2.1 and optionally Equation 2.2 with the values taken at time \( t \), adding any forces from internal collisions or external
sources. Then using this force to solve the time-integrated equations of motion:

\[ v_{t+\Delta t} = v_t + \frac{F_t}{M} \Delta t \]  \hspace{1cm} (2.5)

\[ x_{t+\Delta t} = x_t + v_{t+\Delta t} \Delta t \]  \hspace{1cm} (2.6)

finding the new velocity \( v_{t+\Delta t} \) from the acceleration from the sum of forces \( F_t \) at time \( t \), before finally updating the position to \( x_{t+\Delta t} \) with the new velocity.

Euler’s method of solving these equations is simple to implement and fast to calculate, however, can be highly unstable for stiff systems if the timestep \( \Delta t \) is not small enough. If the stiffness \( k \) in Equation 2.1 is large then any small changes to the length of the springs can create large forces, which in turn can move the mass beyond the point of equilibrium of the spring system and create a larger extension in the other direction. This leads to an exponential increase with each time step. As the time steps for real-time applications such as video games are relatively large, this use of explicit Euler integration as done by [Norton et al., 1991] and [Imagire et al., 2009] is possibly unsuitable.

Instead, an implicit integration scheme might have to be used. To calculate the forces from the springs with the changes of length taken at time \( t + \Delta t \), meaning that the velocity of the masses depends on the forces from the springs, which in turn depends on the unknown position of the masses at time \( t + \Delta t \), and the positions of the masses now depend on the unknown velocities also at time \( t + \Delta t \) [Nealen et al., 2006]. This creates a system of equations to be solved, that can either be solved analytically if the force is linear, through methods such as Gaussian elimination, LU decomposition, or iteratively through methods such as Newton-Raphson, steepest descent, or conjugate gradient [Ferziger and Peric, 2002]. If any of the forces is non-linear then the system of equations is non-linear, meaning that it has to be solved iteratively. The analytical methods once solved provide the exact answer to the system of equations within the bounds of any rounding errors introduced through the calculations, however, they are also far more expensive than the iterative methods, while iterative methods take a guess value of the solution and with each iteration improve the result. As the level of accuracy
of the exact methods is not necessary, it is the visual appearance that is important, paired with the ability to stop the iterative solver when there is not enough time left before the next frame, to keep a consistent framerate [Parker and O’Brien, 2009b], one of the iterative methods would be more appropriate.

A simple Implicit integration of the problem without dampening provides the equations for the entire system:

\[ v_{t+\Delta t} = v_t + \frac{F_{t+\Delta t}}{M} \Delta t \]  

(2.7)

\[ x_{t+\Delta t} = x_t + v_{t+\Delta t} \Delta t \]  

(2.8)

these represent the system of equations for the body, with \( M \) as the sparse matrix of masses, while each of the other variables is a vector containing the vectors for the positions and velocities in each direction for every mass in the system at either time \( t \) or \( t + \Delta t \). To solve this iteratively the problem can be cast as an optimisation problem. [Liu et al., 2013] derives the minimisation problem from Equations 2.7 and 2.8 as the system of equations:

\[ M(X - Y) = \Delta t F_{t+\Delta t} \]  

(2.9)

where \( X = x_{t+\Delta t} \) and \( Y = 2x_t - x_{t-\Delta t} \). The solution corresponds to the critical point of the integrated equation:

\[ g_X = \frac{1}{2}(X - Y)^T M (X - Y) + \Delta t E_X \]  

(2.10)

where \( E \) is the potential energy of the spring:

\[ E = \frac{1}{2} (\|p_1 - p_2\| - r)^2 \]  

(2.11)

found from the extension of the spring. Essentially the minimum of \( g_X \) occurs at \( \nabla g = 0 \) which is the same as Equation 2.9. This turns the minimisation problem into a multidimensional root finding problem of Equation 2.9. Enabling the use of the Newton-Raphson method to find the root. The Newton-Raphson method takes the gradient of
the equation at the point of the guess value to estimate where the root of the equation lies, the gradient at this new value is then found and the process is repeated until the value is close enough to the root. This method is popular for its ability to find the root quickly if the initial guess is relatively close to the root, however, it can have issues with overshooting the root and ending up further from the root than the previous guess, or if it finds a saddle point or local maximum or minimum, it can end up not returning the next guess as the gradient is 0 [Ferziger and Peric, 2002, Liu et al., 2013]. While this solution does converge well when placed near the root, it does require solving the system of equations each iteration, which have to be rebuilt regularly as the system changes preventing the system from being pre-factorised before runtime [Liu et al., 2013]. [Liu et al., 2013] implemented a system to solve this problem more efficiently for use in computer graphics, converting the problem to two stages, a globally linear in the mass positions and locally non-linear in spring directions. As the global linear component does not rely on the runtime state of the mesh, the matrix could be built prior to runtime, allowing for an increase in speed. While their method does have poor convergence properties as compared to the Newton-Raphson method, theirs was proven to be faster in the first few steps, which is critical in real-time applications that do not have the time to run a large number of iterations. Unfortunately, this method is not suitable for projects that involve changes to the system, such as tearing, or fracture, as it requires recalculating of the system matrix, removing the benefit of the method precalculating said matrix.

The problem could be kept as a minimisation problem and open up the solution to iterative solvers such as successive over coordinate direction, steepest descent, or conjugate gradient. All of which aim to minimise the Equation 2.10, by repeatedly minimising the equation in a single search direction before finding a new direction. This results in the solver effectively descending towards the minimum with each new iteration. Each of these has their advantages and disadvantages towards finding the minimum. Successive over coordinate directions is a simple method that locks the search to one axis, once the minimum is found it changes to the next axis and repeats until the final solution is found. This is effective, however, is not the most efficient. Steepest descent picks its search direction as the negative of the local gradient at that point, which to the solver’s knowledge at that point is the quickest way down to the solution. Once it has found the minimum of that search direction, the next search direction will by definition be at
90° to the previous search direction, and this process is repeated until it converges. The disadvantage of this solution is that it can be slow to converge when the equation results in a narrow valley, with solver moving across the valley multiple times before eventually finding the final solution [Ferziger and Peric, 2002]. Conjugate gradient avoids this problem by never searching in the same direction twice. This is achieved through the ability to minimise in two planes at once if the two directions are conjugate. Two directions \( d_1 \) and \( d_2 \) are conjugate with respect to matrix \( A \) if \( d_1 \cdot Ad_2 = 0 \). This allows this method to hone in on the exact solution once the number of iterations reaches the size of the matrix, however, finding the exact solution isn’t necessary [Ferziger and Peric, 2002].

While the mass-spring method is appreciated for its speed as compared to the Finite Element method, the calculation of the movement of every mass in a body through either explicit or implicit means is expensive. This led to the work of [Smith et al., 2008] to avoid the calculating of how each mass responded to the forces from the springs, where the springs were replaced with distance preserving constraints. This allowed for the simulation of brittle fracture through the calculation of the forces required to keep each mass in its original orientation, with the constraint removed once the force on it is raised past a critical value. This proved to speed up the method, however, removed the more realistic elastic behaviour that the method benefited from, entirely. If this elastic behaviour is not of importance, then this approach still allows for the more realistic simulation of where cracks initiate and how they propagate than what is easily achieved through geometric models.

### 2.3.2 Finite Element Method

Finite element method is used in engineering literature to simulate complex mechanical components that cannot be analysed using basic methods. It breaks the component down into discrete elements, typically triangles in 2D and tetrahedrons in 3D, with nodes placed on each corner for both, with each element connected to its neighbours through their shared nodes. The stress can now be calculated across these much simpler elements with forces, displacements, and boundary conditions applied to the nodes of the elements [Budynas and Nisbett, 2011].

Where the mass-spring method could provide a rough approximation of the deformation a material would undergo, with a loose coupling associated with the fracture criteria
and the stress that causes a fracture. Finite element method allows for the accurate
calculation of the stress and deformations that a body undergoes under an applied load
through the governing mechanical equations, meaning actual material properties such as
the yield and tensile strength can be defined for use in determining how a body deforms
or how it fractures.

Finite element method benefits from being able to model the stress and strain accurately
across non-uniform grids, through the use of weight functions applied to the equations.
This is commonly taken advantage of in engineering practice by increasing the resolution
of the mesh in areas where high stress is expected and reducing the resolution further
away. Resulting in a more accurate simulation of the stress profile within the material
when done correctly, without having to increase the simulation time to accommodate the
higher resolution across the entire object. While this may seem similar to the concept
of increasing the mesh resolution around impact locations for geometric or mass-spring
models to get more detail out of the resulting fracture, as, like previous methods, the
basic finite element method is limited to fracture along the boundaries of the elements.
Finite element method, however, calculates the stress across the body of the object to
determine where the fracture will happen, this is most likely to occur in locations of
stress concentrators, such as hard edges or cracks. This could be taken advantage of by
applying a greater resolution mesh around these known stress concentrators to allow for
more detailed fracture in the region it is most likely to occur. Therefore, a non-uniform
mesh in finite element method is more appropriate for video games as a prior knowledge
of impact location is not necessary, rather, it is an idea of where the stress is likely to
build in the objects, which could provide a benefit to the resulting destruction.

[O’Brien and Hodgins, 1999] adapted a finite element method for use in computer graph-
ics, calculating the displacements of all the nodes based on the applied forces and bound-
ary conditions. From this, the strain could be found, as the strain is a measure of change
in length in a material. This links the displacements of the nodes to the resulting inter-
nal forces that in turn generate the stresses throughout the body. With this [O’Brien
and Hodgins, 1999] use the stresses in their separation tensor which allows for the com-
parison of the stress as to the material’s resistance to crack propagation known as its
toughness. In this initial work, this was limited to brittle fracture, however, [O’Brien
et al., 2002] showed that some small adjustments through breaking the strain down
into elastic and plastic components in [O’Brien and Hodgins, 1999] would allow for the
accurate simulation of plastic deformation and thus ductile fracture. However, these methods were not viable for use in real-time applications as they were too slow. [Parker and O’Brien, 2009b] built upon the work of [O’Brien et al., 2002, O’Brien and Hodgins, 1999] optimising it through multithreading, and the use of iterative solvers, to solve the deformations of different chunks of the body in parallel to speed up this method to the point it was used in a commercial video game.

The use of finite element method and tetrahedral elements allows for the dynamic remeshing of the body as the simulation runs to get more accurate results. In [O’Brien et al., 2002, O’Brien and Hodgins, 1999] this was done through the slicing of the tetrahedral element into multiple smaller tetrahedrons along the plane the crack was travelling. In essence, adding elements to the mesh to allow the crack to travel along the directions dictated by the internal forces, instead of having to follow the boundaries of the initial tetrahedral mesh, like in pre-fractured models. However, this does not mean that a relatively high-quality mesh is not required. As cracks propagate they cross over the entire element, meaning if the mesh is too low a resolution it can lead to the crack making large jumps across the material. [Koschier et al., 2014] proposed a method of re-meshing the object around areas of high stress, then when the stress reduces, the mesh is returned to its initial more coarse state. Enabling for the more accurate calculation of the stress and propagation of the cracks without having to permanently increase the memory usage, unlike in [O’Brien et al., 2002, O’Brien and Hodgins, 1999], which provide no way of reducing the number of elements once they have been added. While these methods do allow for more accurate calculation of the stress, or of the representation of the cracks propagation through the material, they do so at the risk of a substantial increase in the number of elements in the simulation that will have an impact on the performance. This could be controlled if the number of elements in an object was capped, though this still leaves the risk that multiple objects in the scene could start increasing the number of elements at the same time. This could require a cap on the number of elements in the scene total or the cap on each individual object to be tuned based on the total limit for the scene. As for [Parker and O’Brien, 2009b] they dealt with the problem by simplifying the fracture to exclude the re-meshing all together, instead choosing to prevent fracture of an object if there are less than three tetrahedrons.

[Parker and O’Brien, 2009b] chose to use lower resolution structural meshes rather than the meshes used, for rendering for the purpose of efficiency. Pairing these structural
meshes with higher detailed textured meshes called splinters, similar to the pre-fractured method. While this may suffer from the same issues as discussed earlier for pre-fractured methods, the finite element component determines which splinters break off the model, as these splinters are linked to specific tetrahedral elements of the structural mesh. This use of splinters results in the destruction that appears to have a higher level of detail than the structural mesh being used and is a concept that could be carried over to the mass-spring method to provide greater detail without the extra cost of a greater resolution.

As with the mass-spring methods systems of equations in finite element method have to be integrated through time to push the simulation forward. This can be done with explicit Euler providing simplicity and speed to the implementation, or, can be done implicitly, achieving a higher stability with the added cost on computation. The arguments for and against each are the same as discussed previously. However, unlike mass-spring methods, implicit integration seems to be the more popular choice due to its improved stability [Parker and O’Brien, 2009b].

2.4 Evaluation

The mass-spring method has also proven to be incredibly flexible, while already capable of simulating the elastic behaviour of a material. Through simple additions, a developer can achieve the simulation of plastic deformation [Terzopoulos and Fleischer, 1988], or model the imperfections that are found within materials [Norton et al., 1991]. While [Imagire et al., 2009] showed that the model could be extended with the inclusion of dust and debris generation to improve upon the visual quality of the destruction created. The results reported by [Imagire et al., 2009] show that however, the dust and debris generation is not yet applicable for real-time use as they are too computationally expensive. The other additions listed will all have an adverse effect on performance, however, this is minimal in comparison to the storage requirements of the system in general. The use of a slip unit to include plastic deformation will require storing the plastic deformation for each spring, this extra piece of data could become substantial for large systems, and may be worth taking into consideration. The same point can be noted for the inclusion of the simulations of imperfections, this would require generating and storing the unique critical lengths for each spring rather than comparing against
a single value for the system. As with the pre-fractured method, this project is going to focus on the basic method using the same pre-produced Voronoi based tetrahedral meshes constructed from NetGen. This will allow for an accurate comparison between the performance of the two methods, as they will be performed on the same tetrahedral meshes. To progress the simulation through time the explicit Euler integration will be applied, as it is the preferred method for mass-spring systems, due to it being less computationally expensive.

The process of qualitatively evaluating these methods is typically done through checking the visual appearance of the results of the simulation, sometimes comparing to photos of the equivalent object in the real world breaking in a similar manner as done by [O’Brien and Hodgins, 1999]. Others simply provide images of the results for the reader to decide if this method provides a look that they wish to achieve, such as in [Imagire et al., 2009, Müller et al., 2013, Müller et al., 2001]. This project is going aims to present the results of the destruction to the participants of a survey and request

While to measure the performance of the object, typically the frames per second (FPS) achieved during the simulation is used [Imagire et al., 2009] or the inverse, the time to simulate and render one frame. As for simulations that do not work in real time like [O’Brien et al., 2002, O’Brien and Hodgins, 1999] the time to simulate a single second is provided. However, these papers do not provide a breakdown of how the performance changes over time during a fracture event, which would be beneficial for comparing static methods such as the pre-fractured method, that only occurs when a significant force is applied, to the mass-spring method, which is continuously simulating the forces occurring within the object, even when the force applied is not large enough to cause fracture.

2.5 Conclusion

Many different approaches to creating destruction through geometric-based methods have been applied throughout the literature covered. The pre-fractured method has proved to be flexible, allowing for the improvements in runtime speed, the memory usage, the authoring time, and the quality of the destruction, as long as the developer is willing to sacrifice some of the other variables. Through the graphical process of instancing,
the memory usage can be reduced but restricts the destruction to multiples of the same fragments which may not represent the object as a whole particularly well. While, the quality of the destruction can be improved by increasing the memory usage and authoring time, through the creation of variant destruction patterns that are chosen at random on impact. The quality could also be improved through higher resolution meshes in areas of stress concentrators which may provide the destruction with the appearance that the higher stress in these areas tore the object into finer pieces. There are many options in using this method for a developer to get the desired effect, however, it is not within the scope of this project to implement each variation of this method to compare their effects on the performance. This project will focus on a basic pre-fractured method, that creates the Voronoi-based fragments on impact, using the NetGen tool as recommended by [O’Brien, 2000] to pre-produce the tetrahedral mesh, as it is not possible to create the Voronoi diagram at runtime [Yan et al., 2010].

As discussed previously, when these methods were evaluated by [Muguercia et al., 2014] the results were based on their performances on the machines they were implemented on in their respective papers. The performance rating on a five-point scale was approximated based on the original results and the hardware the results were achieved on. To improve upon this research, this project aims to provide the performance measured in FPS, or in time to render each frame across the fracture event, to provide a more accurate comparison between the two methods. The comparison in quality has not been quantified previously in the reviewed literature as [Muguercia et al., 2014] looked to provide more information on the solvers used in the different methods to accompany the approximation of the performance. They did not aim to determine if there existed any improvement in the perceived quality from more computationally expensive methods.

The use of the finite element method in computer graphics is a topic of extensive research, it stands above the other available methods for its ability to accurately simulate the stress and deformation of the material. It allows developers to create virtual materials, specifying the material constants used in the calculations to determine how a material reacts to stimuli [O’Brien, 2000]. However, it is more expensive than other methods and requires a significant amount of optimisation before it is viable for real-time use. Due to this expense, this method typically requires coarser meshes to be able to operate quickly, which can result in a button popping effect [O’Brien, 2000] where cracks move unnaturally in large steps through the material. [Parker and O’Brien, 2009b] used a
separate structural mesh to perform the deformation and fracture simulation on and linked each tetrahedron of this structural mesh to points in the original mesh to allow the use of the method with more refined textured meshes. A common problem with previous methods, is the destruction is limited to the boundaries of the fragments, finite element method in its most basic form suffers the same issue, however, it has been extended by [O’Brien and Hodgins, 1999] to split tetrahedrons up on the plane the crack is travelling on, thus shifting the boundaries to where the crack would propagate. While [Koschier et al., 2014] tried to combat the issue by dynamically increasing the resolution of the mesh in areas of high stress. Although both solutions open up the method to go beyond that of pre-fractured and mass-spring, these solutions are both quite expensive. This increase in tetrahedral elements from either solution would require limitations to be put on the number of new elements that could be created, to prevent a significant impact on performance. As with the two previous methods, it is not feasible to explore each of these variations, therefore this project is going to focus on further exploration of the finite element method on the implementation of the work of [Parker and O’Brien, 2009b]. The work of [Parker and O’Brien, 2009b] has proven to be viable for real-time use, and an investigation of its use of an implicit integration scheme would be transferable to the mass-spring method if necessary.
Chapter 3

Methodology

To complete the aim of this project, to compare a number of methods for modelling fractures in computer graphics, for use in real-time applications and to highlight differences in the performance and the quality of the destruction, it is necessary to implement each method. This chapter sets out to detail the environment and tools used to conduct the project, how the pre-fractured and mass-spring methods were implemented, as well as how these methods were evaluated both quantitatively through performance tests and semi-quantitatively through a survey. Before discussing the steps necessary to implement the finite element method.

3.1 Environment and Tools

This section will detail the specific hardware and software used to conduct this project along with the process of generating the tetrahedral meshes and how these meshes were imported into the project, as this process is universal across the different methods discussed.

3.1.1 Hardware

The entirety of the project development and experiments were conducted on a Mid-2014 MacBook Pro running macOS High Sierra, with the hardware specifications:

- Processor - 2.8 GHz Intel Core i5.
• Memory - 8 GB 1600 MHz DDR3.
• Graphics - Intel Iris 1536 MB.

3.1.2 Tools

The models were implemented in Unity version 2017.3 Personal, using Visual Studio Code with all scripts written in C#. This was chosen as it is an industry standard tool with a large amount of documentation available. With the underlying engine proven to be stable, any issues can be safely assumed to be coming from the methods implemented and not with what they are running on. The tetrahedral meshes were generated with NetGen 6.2 from .stl surface meshes created in Blender 2.79.

3.1.3 Generating Tetrahedral Meshes

As discussed in Chapter 2, both methods implemented use a Voronoi based tetrahedral mesh, to be pre-computed using the NetGen tool. From there the tetrahedral mesh is imported from the file into the Unity environment at runtime. The process of taking a surface mesh in .stl format and generating the tetrahedral mesh with NetGen is as follows:

1. Select File > Load Geometry.
2. Select .stl surface mesh file from directory.
3. Optional: Adjust Mesh resolution:
   (a) Select Mesh > Meshing Options.
   (b) Adjust Mesh granularity to change resolution.
4. Choose Generate Mesh.
5. To save generated mesh, select File > Save Mesh.

3.1.4 Loading Tetrahedral Meshes

The outputted file is of the format .vol, consisting of multiple space separated columns, each under a heading detailing what information is stored in the following lines. The
three important headings are "points", "surfaceelements" and "volumeelements". The lines following the heading "points", stores the position in each of the 3 axes for each vertex on a different line. In C# the Split method can be used to separate the line by the empty spaces, in the resulting array of values the positions can be found at indices 0 to 2 in the order x, y and z. Under the heading "surfaceelements", a lot more information for each surface triangle can be found than for the vertices. Again, each triangle is stored on a different line, with each value in the line separated by a space. In the resulting array the indices, for each triangle, are stored at the indices 5 to 7 if the line is separated by the empty spaces. These indices are for the points that make up each triangle and start at 1, therefore, they have to be corrected before any further use by subtracting 1. Below the heading "volumeelements", the information for each tetrahedral element is stored. Each element is on its own line and the values are space separated as before. Once split by the spaces, the indices for the points that make up that tetrahedral element are found at indices 2 to 5, like with the indices for the surface triangles, the indices of the points that make up the tetrahedrons start at 1 and have to be corrected before any further use.

Algorithm 1 Read .vol file

1: for each line do
2:   if line is "points" then
3:     Set mode to points
4:   else if line is "surfaceelements" then
5:     Set mode to surface
6:   else if line is "volumeelements" then
7:     Set mode to volume
8:   else
9:     if mode is points then
10:        Split string and add data at positions 0, 1 and 2 to vertices
11:     else if mode is surfaceelements then
12:        Split string and add data at positions 5, 6 and 7 to triangles
13:        Subtract 1 from data, to correct indices to start at 0
14:     else if mode is volumeelements then
15:        Split string and add data at positions 2, 3, 4 and 5 to elements
16:        Subtract 1 from data, to correct indices to start at 0

To integrate this into the experiments a parser class was written following Algorithm 1 to open the file specified in the Unity inspector, search for each of the headings listed and add the corresponding data to their respective lists of Vector3, int and Vector4. This
allows for the lists to be outputs from the parser object to the required Unity GameObject. With the vertex list and surface triangle indices passed to the GameObject’s Mesh object as detailed in [Unity Technologies, 2018c], to programmatically create the objects mesh.

With the process of importing the tetrahedral meshes into Unity now described, it is possible to move on to how to implement the individual methods to be tested.

3.2 Methods

This section details the implementation of the two methods, pre-fractured and mass-spring, that were implemented and compared in terms of performance and quality of the destruction they produce. The discussion on how the implementation of the finite element method can be achieved is found in Chapter 3.4.

3.2.1 Pre-Fractured

The pre-fractured method is a static geometry-based method. That uses a surface mesh to represent the object until that object is impacted with a force greater than the critical value of the object. At this point, the object is simply replaced with its fragments. In this case, these fragments are the tetrahedrons from the tetrahedral mesh.

Figure 3.1: Example of implemented pre-fractured destruction, illustrating a simple shape collapsing
Chapter 3. Methodology

<table>
<thead>
<tr>
<th>Prefractured</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ meshFile : string</td>
</tr>
<tr>
<td>+ criticalForce : float</td>
</tr>
<tr>
<td>- elements : List&lt;Vector4&gt;</td>
</tr>
<tr>
<td>- Start() : void</td>
</tr>
<tr>
<td>- OnCollisionEnter(Collision) : void</td>
</tr>
<tr>
<td>- CreateInitialMesh() : void</td>
</tr>
<tr>
<td>- fracture() : void</td>
</tr>
</tbody>
</table>

**Figure 3.2:** UML diagram for the Prefractured class.

The implementation of this method in Unity, an example of which is shown in Figure 3.1, was done by creating an empty 3D GameObject object, providing it with the basic component objects necessary for a 3D object in Unity:

- A MeshRenderer.
- A MeshFilter, without a mesh as this will be added programmatically.
- A Rigidbody.
- A MeshCollider.

A pre-fractured script was created for this GameObject shown in Figure 3.2, to handle applying the surface mesh on start, generating the fragments upon impact, and removing the original object once fracture is completed.

**Algorithm 2** Initialise Prefractured object

1: Get vertices, triangles indices, and element indices from file
2: Store element indices for later use
3: Assign vertices to the mesh
4: Assign surface triangle indices
5: Calculate surface normals - Note: can use mesh.RecalculateNormals() in Unity
6: Provide collider the surface mesh

On start, the initial surface mesh has to be created, the vertices, surface triangles and tetrahedral elements are all read from the .vol file as described in Chapter 3.1. The process of creating the surface mesh as shown in Algorithm 2 is as simple as assigning the arrays of vertices and surface triangles to the MeshFilter’s mesh.vertices and mesh.triangles variables as detailed in [Unity Technologies, 2018c]. To allow for collisions with any other objects in the scene, the Mesh Collider attached to this object has to be given a mesh, this is completed through assigning the MeshFilter’s mesh to the
MeshCollider’s sharedMesh variable. As the fragments are not created until there is an impact with a great enough force, the indices for the tetrahedral elements must be stored for later use when performing the fracture operation.

**Algorithm 3** Generate fragments for pre-fractured object

1: if Collision starts then
2: Calculate total force acting on object
3: if Force ≥ critical then
4: for Each tetrahedral element do
5: Generate fragment object
6: Use indices of the tetrahedral element to get vertices for fragment
7: Assign vertices to fragment mesh
8: Assign surface triangle indices to fragment mesh - Note: For a tetrahedron surface triangle indices were found to be \{0, 1, 2, 1, 3, 2, 0, 2, 3, 0, 3, 1\}
9: Calculate surface normals - Note: can use mesh.RecalculateNormals()
10: Provide Collider the surface mesh of fragment
11: Assign fragments position and rotation to that of the original object
12: Assign velocity, rotational velocity and mass
13: Optional: Set fragment to be destroyed in a set number of seconds to prevent scene becoming cluttered
14: Destroy Original pre-fractured object, it is no longer needed

The process of fragmenting the object on impact is shown in Algorithm 3. It is achieved by taking advantage of Unity’s built-in OnCollisionEnter method. This takes a Collision object as an input, that contains information on the object that was hit, the total impulse of the collision and the relative velocity [Unity Technologies, 2018a]. With the impulse, the magnitude of the total force acting on the object can be calculated with:

\[
F_{total} = \| I / \Delta t_{fixed} \|
\]  

(3.1)

where \( I \) is the total impulse and \( \Delta t \) is the time at which this impulse was applied in this case the time from the last Unity FixedUpdate obtained through the Time.fixedDeltaTime variable. With the magnitude of the total force calculated this can be compared with the critical force of the object. If the total force is greater, then the fracture can be initiated. Each fragment for the fracture is generated in a for loop, in each iteration, a new GameObject is created and provided the same essentials as for the original object, a MeshRenderer, MeshFilter, Rigidbody and a MeshCollider. The important information is assigned for each of these objects before moving on to the next fragment. To create
the mesh for the fragment, the indices corresponding to the vertices of each tetrahedral element are used to retrieve the vertices to be assigned to the mesh.vertices variable for the fragment. As for the surface triangles, these are the constant for every tetrahedral element in the system, therefore a constant array is passed to mesh.trianlges. Finally, the appropriate positions, rotations and velocities are set for the transform and rigid body of the fragment before moving on to the next tetrahedral element. Once every tetrahedral element has been generated, the original pre-fractured object is no longer required in the scene and the Destroy method can be used to call for it to be disposed of by the garbage collector.

This setup allows for the pre-fractured object to be added to the scene using Unity’s prefab pattern requiring only a filename for the mesh and a critical force for when fracture should occur to be given.

3.2.2 Mass-Spring

The mass-spring method is a physically-based method, connecting each of the elements, that form the object as a whole, with springs. As forces are applied to the object, it moves the fragments stretching or compressing the attached springs, in turn, this generates forces to restore equilibrium to the system. This allows for the system to mimic elastic deformation during collisions, and through removing springs, this method simulates the fracture. The springs in the system have a critical length ratio. Once a spring is stretched past this point, the spring is removed stopping it from being able to provide that restoring force holding the object together.

For the implementation of this method an example of the results are shown in Figure 3.3, the system was split into multiple parts, the massSpringSystem GameObject, the element GameObject and a spring object as shown Figure 3.4. The element GameObject is for each individual tetrahedral element in the system and just like the pre-fractured fragments discussed in Chapter 3.2.1 requires the MeshRenderer, MeshFilter, Rigidbody and MeshCollider objects to be visible and have a physical presence in the scene. Spring objects connect each element to each of its neighbours and handle the calculation of the restoration force to act on the two elements connected. While the massSpringSystem GameObject controls the entire system for that physical object through the attached script, loading the mesh data, generating the elements, instructing the springs when to
Chapter 3. Methodology

Figure 3.3: Example of implemented mass-spring destruction, showing the frames from a collision event

<table>
<thead>
<tr>
<th>MassSpring</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ meshFile : String</td>
</tr>
<tr>
<td>+ elementPrefab : GameObject</td>
</tr>
<tr>
<td>+ numberOfCommonVertices : int</td>
</tr>
<tr>
<td>+ massSpringLengthRatio : float</td>
</tr>
<tr>
<td>+ stiffness : float</td>
</tr>
<tr>
<td>- elementIndices : List&lt;Vector4&gt;</td>
</tr>
<tr>
<td>- springs : List&lt;Spring&gt;</td>
</tr>
<tr>
<td>- elements : List&lt;Element&gt;</td>
</tr>
</tbody>
</table>

- Start() : void
- FixedUpdate() : void
- CreateElements() : void
- CreateSprings() : void

<table>
<thead>
<tr>
<th>Spring</th>
</tr>
</thead>
<tbody>
<tr>
<td>- stiffness : float</td>
</tr>
<tr>
<td>- elements : element[]</td>
</tr>
<tr>
<td>- restLength : float</td>
</tr>
<tr>
<td>- broken : bool</td>
</tr>
</tbody>
</table>

+ Spring(Element A, Element B, float stiffness)
+ getLengthRatio() : float
+ calculateRestorationForce() : void
+ breakSpring() : void
+ isBroken() : bool

Figure 3.4: UML diagram for the MassSpring class.

calculate restoring forces, and removing springs once they have reached their critical length and are classed as broken.

To use a single surface mesh would require calculating which vertices are on the outside of the volume everytime a spring was to break to determine whether or how the singular surface mesh had changed. Instead, in this implementation of the mass-spring method, the object was built from each individual element with its own surface mesh connected by springs from the start. Therefore, the process of creating the object in the scene at the start follows a similar process to Algorithm 3. In this case, omitting the removal of
Algorithm 4 Generate mass-spring object

1: Generate fragments, as done in Algorithm 2
2: for element $i = 0$ to element $i = N$ do
3:     Check if element $i$ connected to element $j$
4:     for element $j = i + 1$ to element $j = N$ do
5:         for Each node of $i$ do
6:             for Each node of $j$ do
7:                 if Node of $i$ matches node of $j$ then
8:                     Add to count
9:                 Break from loop
10:            if count $\geq$ required then
11:                Create new spring, pass reference to element $i$ and $j$
12:                Add spring to the list of springs in system

the original object, once completed the springs have to be added to the system. The
process for this is shown in Algorithm 4. For every unique pair of elements, it checks
the number of nodes that the two elements share, if this is greater than or equal to
the requirement specified a spring is added. The required number of shared nodes is
between one to three: A single shared node means the two elements are connected by
a single vertex, two shared nodes means the elements are connected along an edge and
three shared nodes mean they are connected by a single face. Decreasing the number
of required shared nodes, significantly impacts the performance of the method as it can
lead to a large increase in the number of springs. However, it will improve the stability
of the method by thus allowing for the object to be better held together. It was decided
to perform the search for connecting pairs, on start for each model to allow for the type
of connection between pairs that determined if a spring would be added to be variable.
Allowing for more or fewer springs to be added quickly, based on the needs for that
object. This process could easily be shifted over to a separate program, and become
part of the pre-production workflow for the models. A change over to pre-production for
generating springs would require saving the details on which elements were connected
by springs to a file, and for this to be read at the start of each model along with the
tetrahedral mesh. When creating each spring, references to the two connected elements
are passed, this allows for the spring to retrieve its rest length as the magnitude of the
difference between the positions, calculate the deformed length at runtime, and based
off the difference, being able to calculate and apply the restoring forces.
To update the massSpringSystem, it runs through each spring in the system, checking if its current length ratio, calculated using:

\[ l_{ratio} = \frac{\|x_2 - x_1\|}{l_{rest}} \] (3.2)

is less than its critical length ratio, where \( x \) represents the position of its respective element. If true, the restoring force can be calculated using:

\[ F = k(\|\Delta x\| - l_{rest}) \ast (\Delta x/\|\Delta x\|) \] (3.3)

and applied to the two elements. In Equation 3.3, \( k \) is the stiffness of the spring, \( \Delta x \) is the current length of the spring and \( l_{rest} \) is the spring's rest length. The \( (\Delta x/\|\Delta x\|) \) component of Equation 3.3 provides the direction for the force vector. When applying this force vector to the two elements, element 1 is provided the force \( F \) while element 2 is provided the negative, force \( -F \), this way the forces acting on both elements are going outwards when the spring is compressed and inwards when stretched. It should be noted that care should be taken when applying the restoring forces to the elements, only once the restoring force has been calculated for every spring should the forces be applied, prevent elements moving between forces being added by different springs. In Unity the forces are added through the AddForce method of the Rigidbody, taking advantage of Unity’s built-in physics engine to move the individual elements after the FixedUpdate method, which instructs each spring to calculate its restoring force, has been concluded.

If the current length ratio is greater than the critical length ratio, then the spring is removed from the system, thus performing a single fracture event for the object.

This orientation enables the object to be added to the scene by adding the massSpringSystem prefab to the scene. The object is physically placed in the scene according to the orientation of this massSpringSystem GameObject. The key information for the system, such as the name of the mesh file, spring stiffness, and spring critical length are all provided through the inspector for the massSpringSystem GameObject.
Chapter 3. Methodology

3.3 Evaluation

The process of comparing the two methods, pre-fractured and mass-spring, was completed through a quantitative analysis of the performance of the two methods and a semi-quantitative analysis of the quality of the destruction created. To determine if there is a difference in performance between the two methods and if there exists a perceived improvement from using one method over the other.

3.3.1 Performance

The quantitative analysis of the performance of the two methods was achieved by recording the time to render each frame across a fracture event. From this data, significance testing was performed on the means to prove false the null hypothesis, that there exists no difference in how fast the two methods perform.

To test the performance of both methods, a sphere was fired at the destructible object to cause a fracture event. From the creation of the destructible object and the ball, the time to render each frame was recorded by writing the value of Time.DeltaTime to a space separated file. The time to render each frame was chosen instead of the time spent to complete different methods, as the mass-spring method makes use of the physics engine built-in to Unity, which the time to render each frame takes into account, thus providing more accurate information of the performance of the two methods. The reason why Unity’s built-in profiler was not used to record the data, is although, Unity’s profiler records the time to process each of the individual components; physics, scripts and rendering for each frame, the profiler does not make this information available outwith the profiler environment to perform statistical analysis on the collected data, and therefore, is not suitable for fulfilling the goals of this project. While writing to file does have an impact on performance, as this is universal across every experiment for both methods it can be assumed that this will not affect the comparison of the two methods, as the same amount of information is being recorded each frame for both methods.

The experiment was set up using an ico mesh from blender for both destructible objects. Using Netgen, a moderate size tetrahedral mesh was generated of 198 elements. The scene created in Unity consisted of a flat terrain object set to a y-position of 0 m to
Table 3.1: Summary of variables for performance experiment

<table>
<thead>
<tr>
<th>Method</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both</td>
<td>Number of tetrahedral elements</td>
<td>198</td>
</tr>
<tr>
<td>Both</td>
<td>Position of object</td>
<td>(0, 0.5, 0) m</td>
</tr>
<tr>
<td>Both</td>
<td>Position of sphere</td>
<td>(-20, 10, -20) m</td>
</tr>
<tr>
<td>Both</td>
<td>Radius of sphere</td>
<td>2 m</td>
</tr>
<tr>
<td>Both</td>
<td>Mass of sphere</td>
<td>25 kg</td>
</tr>
<tr>
<td>Both</td>
<td>Force applied to sphere</td>
<td>800 N</td>
</tr>
<tr>
<td>Pre-fractured</td>
<td>Critical force</td>
<td>800 N</td>
</tr>
<tr>
<td>Mass-spring</td>
<td>Stiffness</td>
<td>140 N/m</td>
</tr>
<tr>
<td>Mass-spring</td>
<td>critical spring length ratio</td>
<td>1.4</td>
</tr>
<tr>
<td>Mass-spring</td>
<td>Number of common elements</td>
<td>2</td>
</tr>
</tbody>
</table>

act as the floor, with the destructible object placed on top at the position (0,0.5,0). The sphere to be thrown against the destructible object was given a mass of 25 kg, a radius of 2 m, and the use gravity option of its corresponding RigidBody was disabled to make sure the sphere would hit the destructible object in a predictable pattern that was repeatable. The sphere was placed at the position (-20,10,-20) within the scene and to throw the ball, an impulse force was added to the sphere with the value 800 N in the vector direction towards the centre of the destructible object.

For each experiment, the variables for the destructible objects were kept constant. For the pre-fractured object the value of the critical force was set to 800 N, at this value the object would break when impacted by the sphere thrown at it. While the mass-spring constants were set, a spring stiffness of 140 N/m, a spring critical length ratio of 1.4, and springs were added if two tetrahedral elements shared two vertices, in essence, if they shared an edge, this generated 1322 springs in the system. These constants for the mass-spring system were chosen as this created an object that was reasonably stiff and held together well with a minimum number of springs. As discussed in Chapter 2, the use of explicit time integration with mass-springs systems can suffer from stability issues when dealing with stiff systems, therefore, there does exist a limit to how high the value of the springs stiffness can be set, however, the value of 140 N/m for this particular system was low enough to avoid any stability problems. A summary of the experiment variables can be found in Table 3.1.
To repeat the experiment, a script was written to clear the scene and generate a new sphere and destructible object at set intervals of time. With each new experiment, the data was recorded on a new line on file to separate each experiment. This process was repeated one hundred times to provide a substantial data set to be analysed and to avoid errors in the dataset.

To analyse the dataset, the mean time to render each frame was calculated for each frame, providing the processing time for each frame from when the objects are created at the start of the experiment through the fracture event. To test the statistical significance of the results, the t-test statistical test was used to determine if the null hypothesis, that there is no difference between the performance of the two methods, is false.

The change in time to render each frame across the fracture events for both the pre-fractured and mass-springs methods are reported in Chapter 4, as well as the result of the t-test and the corresponding effect size.

3.3.2 Quality

The semi-quantitative analysis of the two methods was performed through an online survey, where each respondent was shown a clip of each method in action. Each respondent was then asked some simple questions to gauge how realistic they perceived each method to be, and which of the two methods they preferred.

3.3.3 Survey

The survey was constructed and distributed through the online service, SurveyMonkey. Before proceeding with the survey all participants were instructed on what they would be shown in the two clips, what they would be asked, that no personal information would be requested of participants and that responses were anonymised. To start the survey, participants had to agree that they had read and understood the preceding instructions and consented to take part in the research.

The goal of the survey was to obtain semi-quantitative data on the perceived quality of the destruction resulting from the use of the two different methods. To achieve this,
participants were shown a clip of the pre-fractured method and asked to rate on a 7-point Likert scale if they agreed or disagreed with the statement "The destruction of this object looks realistic". The Likert scale was marked: 1 as "Strongly disagree", 4 as "Neither agree or disagree" and finally 7 as "Strongly agree". The second question asked of the participants was "Why does the destruction look realistic? (Tick all that apply or none if unrealistic)", participants were given three options: "The object breaks into pieces as expected", "The fragments of the object move as expected", and "The fracture of the object occurs where expected", as well as a comment box for providing any other additional reasons. These two questions were then repeated for the mass-spring method on the next page of the survey. On the last page, the fifth question asked "Which of the two methods do you prefer the look of" and the final question was "Please explain your answer to Question 5".

It was chosen to ask each participant to what extent they agreed the destruction resulting from the two methods was realistic, rather than asking outright which of the two methods they perceived to be more realistic to avoid adding bias to the question. This approach also allowed for a quantified comparison between the ratings of the two methods that could be tested for significance. As for the second question, this was initially going to be left open-ended to avoid bias, however, after discussing the question with participants of an initial survey, it was revealed that users might agree that a method looked realistic, however, they found it difficult to describe why and wished to see possible examples of what to look for in realistic destruction.

To analyse the data from the survey, the t-test was used on the mean values of ordinal Likert scale from questions one and three, to determine if the null hypothesis that there is no perceived improvement from one method to another, was false.

3.3.4 Legal and Ethical Issues

This section states the legal and ethical issues regarding the use of the survey. While the survey does require human input, no sensitive data was collected on the participants, only how they rate the two methods, their opinions on which they preferred, and on why the destruction looked realistic or not. Participants were only required to watch two short clips under 30 s each and were not at risk of more harm than they would be in a typical office environment. The process falls under the interface only screening, as per,
all participants were over the age of 16 and were either staff or students at Heriot-Watt University. The survey did not require any specialised equipment to complete and could be completed on the participant’s mobile phones laptop, desktop, or tablet. Finally, before being able to proceed with the survey, all participants were required to agree that they had read the instructions for the survey, that they understood why this information was being collected, and to consent to their answers being used for the research purposes of this project.

3.4 Finite Element Method

The finite element method is a process that allows for the simulation of the stresses, and deformations occurring within a complex body that cannot be achieved using the basic stress analysis methods. It does this by discretising the object, typically into tetrahedrons, allowing for the forces, stresses and deformations to be calculated across these much simpler elements. [O’Brien and Hodgins, 1999] used a simplified version of this method for computer graphics, as the accuracy required in engineering is not required in computer graphics, it is the visual appearance that is critical. This section looks to detail the process of how to implement the core elements of the finite element method within Unity, specifically following the work of [Parker and O’Brien, 2009b].

3.4.1 The Mathematical Method

The method revolves around updating the positions of each of the vertices on the tetrahedral mesh, simulating the deformation of the object, in response to any external forces applied. In the process of calculating the deformations, the stresses and forces internal to the object are calculated, which are used to determine if a crack is created and if it will propagate.

The motion of each of the vertices within the object is described through the equation:

\[Ma + Cv + K(x - u) = f_{ext}\]  \hspace{1cm} (3.4)

where \(a\), \(v\) and \(x\) are vectors containing the acceleration, velocity and position for each vertex; \(u\) is the undeformed position of each vertex and \(f_{ext}\) is the external force acting
on each vertex. While $M$, $C$ and $K$ are $N \times N$ matrices corresponding to the mass, dampening and stiffness of the system where $N$ is the number of vertices in the system. The mass matrix $M$ has non-zero values along the diagonal filled with the mass of the corresponding vertices. The stiffness matrix $K$ is constructed of the jacobians of the internal forces on each vertex with respect to the position of another vertex in the same tetrahedral element, calculated using Equation 3.13. With edges shared across multiple tetrahedral elements, the non-zero values of the stiffness matrix are taken as the sum of the jacobians for that position. So if vertex $i$ is in the same tetrahedral element as vertex $j$, then the value at $K_{ij}$ would equal to the sum of the jacobians $J_{ij}$ for each element that shares the edge $ij$. As for the dampening, [Parker and O’Brien, 2009b] use Rayleigh dampening, taking the matrix $C$ as a linear combination of the mass matrix and the stiffness matrix.

To accomplish moving the simulation forward in time, Equation 3.4 is integrated using linearised implicit Euler integration to obtain:

$$
(M + \Delta t C + \Delta t^2 K)v^+ = \Delta t f_{ext} + Mv - \Delta t K(x - u)
$$

(3.5)

where $v^+$ is the new velocity at the end of the timestep. Once solved for $v^+$, this new velocity can be used to calculate the new positions $x^+$ of each of the vertices with:

$$
x^+ = x + \Delta tv^+
$$

(3.6)

Of the variables that compose Equation 3.5, the unknowns are $v^+$, $K$ and $C$. $v^+$ as discussed, is found through solving Equation 3.5, $C$ can be taken as the sum of the mass matrix and the stiffness matrix, while the stiffness matrix $K$ as discussed is built from the jacobians of the force with respect to the positions of other vertices. To obtain the stiffness matrix for the system it is necessary to calculate the following in the order stated:

1. The strain for each element.
2. The stress for each element.
3. The elastic force acting on each node.
4. The Jacobian for each node with respect to every other node in the element.
5. Build the stiffness matrix from the jacobians.

The goal of solving the Equations 3.5 and 3.6 is to obtain the new position of each of the vertices, however, due to the use of implicit integration the new positions are required to calculate the strain to solve the equations. This means that the equations have to be solved iteratively, [Parker and O’Brien, 2009b] chose to use the conjugate gradient method as described in Appendix A.1 until the relative error between iterations is reduced below 0.001 or the max number of iterations had been reached. This way either an accurate approximation of the new position can be calculated or the solver can provide a solution when it hits the maximum number of iterations allowing for the game to maintain a steady framerate [Parker and O’Brien, 2009a].

The first step in building the stiffness matrix is calculating the strain \( \tilde{\epsilon} \) for each element in the system. To do so it is necessary to calculate the deformation gradient \( F \). For each vertex in the element, its undeformed position or reference position \( u_i \) is a column vector, using these the matrix \( D_u \) can be calculated with:

\[
D_u = \{u_2 - u_1, u_3 - u_1, u_4 - u_1\} \tag{3.7}
\]

the same can be done to find the matrix \( D_x \) for the deformed positions. To obtain the deformation gradient the inverse of \( D_u \), \( \beta \) is needed. The process for calculating the inverse of a matrix is detailed in Appendix A.2. Using the equation below the deformation gradient for that element can be calculated:

\[
F = \frac{\delta x}{\delta u} = D_x\beta \tag{3.8}
\]

This deformation gradient, however, needs to be modified before it can be used any further. The method [Parker and O’Brien, 2009b] use for calculating strain is the Cauchy infinitesimal strain tensor, which scales linearly with deformation. The issue is that the strain is affected by the rotation of the element. [Parker and O’Brien, 2009b] use a corotational method that removes the rotation from the calculations of the strain and stress for each element before being factored back in when calculating the elastic force. To remove the rotation from the deformation gradient, polar decomposition is used as described in Appendix A.3 to split \( F \) into an orthonormal component and a symmetric component, to perform the polar decomposition the eigenvectors of the
matrix are required, these are found iteratively using QR decomposition detailed in Appendix A.4. The polar decomposition splits the matrix $F$ into two separate matrices, the orthonormal component $Q$ and the symmetric component $A$. The orthonormal component $Q$ can be used to factor out the rotation from $F$ and does so when calculating the strain $\tilde{\epsilon}$ as follows:

$$\tilde{\epsilon} = \frac{1}{2}(Q^T F + (Q^T F)^T) - I$$

(3.9)

where the superscript $^T$ denotes the transpose of the matrix and $I$ is a $3 \times 3$ identity matrix.

Next, the internal stress $\sigma$ acting on each element in the system is calculated through its relation to the strain with the equation:

$$\sigma = \lambda Tr(\tilde{\epsilon})I + 2\mu \tilde{\epsilon}$$

(3.10)

where $\lambda$ and $\mu$ are material constants and $Tr(\cdot)$ refers to the trace of the matrix, the sum of elements on the diagonal of the matrix:

$$Tr(A) = \sum_{i=1}^{N} A_{ii}$$

(3.11)

The forces acting on each of the vertices within the tetrahedral mesh are related to these internal stresses acting on each of their corresponding elements. The force $f_i$ acting on the node $i$ is equal to the stress times the area the stress is acting on. This translates in the case of the vertex of a tetrahedral element, to the stress acting on the element times by the area of the face opposite that vertex on the tetrahedral element. As this force is a vector it also requires a direction, which is in the direction of the outward facing normal of the opposite face. Finally, before calculating the force, the rotation that was removed earlier through polar decomposition must be factored back into the calculations. This all results in the following equation:

$$f_i = Q \sigma n_i$$

(3.12)

where $f_i$ is the force acting on node $i$, $Q$ is the orthonormal component of the deformation gradient $F$ calculated earlier through polar decomposition and $n_i$ is the area-weighted
normal of the opposite face to vertex $i$, the process of calculating $n_i$ is done through using the vertices of the opposing triangular face to find the base and height of the triangle, and to multiply its area with the normal of that face. The forces acting on each of these nodes serve a dual purpose they are used in calculating the jacobians used to construct the stiffness matrix $K$ and they are used in the calculation of the separation tensor from [O’Brien and Hodgins, 1999].

The final step before constructing the stiffness matrix is calculating the 16 $3 \times 3$ jacobians for each element. There are only 10 unique jacobians, as [Parker and O’Brien, 2009b] notes that $J_{ij} = J_{ji}^T$, therefore it is not necessary to calculate all 16, however it does make the process of constructing the stiffness matrix simpler. The Jacobian for each vertex $i$ of an element with respect to the position of vertex $j$ is calculated with:

$$J_{ij} = -Q\left(\lambda n_i n_j^T + \mu (n_i \cdot n_j)I + \mu n_j n_i^T\right)Q^T$$  \hspace{1cm} (3.13)

where all variables are the same as in previous equations detailed above. In Equation 3.13, the area-weighted normals are column vectors, making the $n_i \cdot n_j$ the dot product resulting in a scalar value that is multiplied by the identity matrix to attain a matrix, while $n_i n_j^T$ is the outer product of the two vectors that results in a $3 \times 3$ matrix.

Finally, the stiffness matrix is constructed by adding each Jacobian to its corresponding element in the stiffness matrix. Any elements on the stiffness matrix that correspond to two vertices that are not connected by a tetrahedral element remain zero. With the stiffness matrix built, the conjugate gradient method can be used to solve Equation 3.5 to find the next value of $\nu^+$ and thus find $x^+$. With each iteration, the process to build the stiffness matrix described above is repeated, until the relative error is below the required level or the maximum number of iterations has been reached. Once this has been completed the separation tensor can be calculated following the process outlined by [O’Brien and Hodgins, 1999].

The fracture algorithm from [O’Brien and Hodgins, 1999] is used to determine if the internal stresses are large enough to create or propagate a crack. With the internal stresses calculated, it is necessary to split these into their compressive and tensile components. This can be accomplished by constructing a symmetric matrix $m(a)$ where one eigenvalue is $\|a\|$, the magnitude of the vector $a$, the other two eigenvalues are 0 and
the corresponding eigenvector for the non-zero eigenvalue is the vector $a$. This matrix is constructed using:

$$m(a) = \begin{cases} 
\frac{aa^T}{|a|} & \text{such that } a \neq 0; \\
0 & \text{such that } a = 0;
\end{cases} \quad (3.14)$$

where once again, $aa^T$ is the outer product of the column vector $a$ resulting in a matrix. This is used to decompose the stress into its tensile and compressive components in the equations:

$$\sigma^+ = \sum_{i=1}^{3} \max(0, v_i(\sigma))m(\hat{n}_i(\sigma)) \quad (3.15)$$

$$\sigma^- = \sum_{i=1}^{3} \min(0, v_i(\sigma))m(\hat{n}_i(\sigma)) \quad (3.16)$$

where $v_i(\sigma)$ refers to the $i^{th}$ eigenvalue of the matrix $\sigma$ and $\hat{n}_i(\sigma)$ refers to the $i^{th}$ normalised eigenvector of the matrix $\sigma$. To get the tensile force $f^+$ and the compressive force $f^-$ exerted by the element on the node, these tensile and compressive stresses are subbed into Equation 3.12 in the place of $\sigma$. This leads to each node having a number of tensile and compressive forces acting on it from all of the elements it is connected to. [O’Brien and Hodgins, 1999] refers to these collections of tensile and compressive forces from each element on a node as $\{f^+\}$ and $\{f^-\}$ and sums each set to obtain the total unbalanced forces for each $f_{\text{total}}^+$ and $f_{\text{total}}^-$. The separation tensor for each node is calculated using the same technique described in Equation 3.14, this time on the vectors of each of the forces.

$$\zeta = \frac{1}{2} \left( -m(f_{\text{total}}^+) + \sum_{f \in \{f^+\}} m(f) + m(f_{\text{total}}^-) - \sum_{f \in \{f^-\}} m(f) \right) \quad (3.17)$$

Using Equation 3.17 allows for direct computation of if fracture is initiated or propagated, through the eigenvalues of the seperation tensor $\zeta$. If the largest positive eigenvalue is greater than the material’s toughness $\tau$ then the node fails and the crack grows. The plane the crack grows in is the perpendicular to the corresponding eigenvector of the largest positive eigenvalue of $\zeta$. Once a node has been failed through checking the
separation tensor, and the plane the crack will grow in, has been found, the node that has failed is copied. Each of the elements connected to the original node has to be assigned to either the original or the new node. To determine which of the two nodes each element is now connected to is completed through checking which side of the crack plane the element lies on. Using a half space test to check the sign of the displacement from the plane to each elements centre, a negative sign shows the element is behind the plane, and a positive sign shows that it is in front of the plane.

This section has covered the mathematical model behind the finite element method, with these equations and processes it is possible to simulate the deformation of an object in reaction to external forces, as well as to calculate when cracks will initiate and if they will continue to grow. Chapter 3.4.2 looks to discuss the specifics of implementing these mathematical equations in Unity and the steps that are necessary to make this method work in real-time.

3.4.2 Implementation in Unity

The implementation of this method in Unity follows a similar process to the previous two methods at first. The vertices, triangles and tetrahedral elements are loaded in and the surface mesh is generated just as was done for the pre-fractured method in Chapter 3.2.1. As for processing the deformation and fracture, this can be done through the calculations discussed in Chapter 3.4.1, taking advantage of the parallel processing capabilities of the graphics card.

To implement this method to run in real-time requires a large amount of optimisation. [Parker and O’Brien, 2009b] created a multi-threaded solution that broke the object into islands, with each thread looping through all the elements found within that island. This allowed for the method to be used successfully in real-time in a commercial video-game. They suggest in [Parker and O’Brien, 2009a] that the future of the method is in parallel processing through the graphics card, and with modern graphics cards containing over a 1000 cores, the deformations of large numbers of vertices can easily be simulated at the same time.
To process these calculations effectively it is best to sort the calculations into sections that are parallelisable. The areas that would best take advantage of this are the calculations from finding the strain to the finding the jacobians, as they are the same for each element; the row-vector calculations when solving the equation of motion, Equation 3.5, for use with the conjugate gradient method; finally the calculation of the results of the stress tensor to discover if a vertex will fail as this is the same process for each vertex in the system.

The first section, from calculating the strain to calculating the jacobians is a simple process of passing the input variables to the shader and performing the calculations within. However, to perform the row-vector calculations involving the stiffness matrix requires the sparse matrix to be stored in a special container. To build a $N \times N$ sparse matrix and to pass that between the graphics card and memory is highly inefficient in terms of processing time and memory consumption, while to include the zero elements in any of the calculations would not add anything to the result while taking up significant time. This problem requires the program store the sparse matrix in a format that only stores the non-zero values and allows for fast performance when calculating row-vector multiplications. The solution is the compressed sparse row list [Pissanetzky, 1984], a description of the format can be found in Appendix A.5. It is not possible to construct this compressed sparse row list straight from the shader calculations, instead of when calculating the jacobians these are placed in a coordinate list format described in Appendix A.6. This coordinate list can then be sorted and the jacobians that have the same row and column can be summed to generate the compressed sparse row list suitable for parallel row-vector calculations.

Finally to determine if any node is to fail through the use of the stress tensor is less clear on how to parallelise, as the calculation of the stress tensor is done for each node, however, the tensile and compressive forces are required from each element for each node. This could be performed as two different dispatches serially to the shaders, first finding all of the tensile and compressive forces from each element on each of its nodes; or these tensile and compressive forces from each element could be calculated in the same shader that produces the jacobians, while this would suffer from repeated calculation of the tensile and compressive forces when iteratively solving the system, it might save time on moving the data back and forth from the CPU and the GPU.
Algorithm 5 Solving Equation 3.5 for use in conjugate gradient analysis

1: for each element, performed within shader do
2: Calculate $D_u, D_x$ using Equation 3.7
3: Calculate the deformation gradient $F$ using Equation 3.8
4: Perform polar decomposition of $F$ to obtain $Q$ and $A$
5: Calculate the corotational strain using Equation 3.9
6: Calculate the internal stress using Equation 3.10
7: Calculate the elastic node forces using Equation 3.12
8: Calculate the jacobians using Equation 3.13
9: Build the stiffness matrix coordinate list
10: Return the coordinate list
11: Sort and sum the coordinate list to build the compressed sparse row list form of the stiffness matrix
12: for each row of Equation 3.5, performed within shader do
13: Perform row-vector multiplications and sum to obtain $Ax = b$ format for conjugate gradient analysis

Algorithm 6 Finite Element Method

1: Generate fragments, as done in Algorithm 2
2: for each frame do
3: while relative error > 0.001 and iterations < max do
4: Perform Conjugate Gradient analysis of output to Algorithm 5 to solve for $v^+$ and update $x^+$
5: Using shader calculate stress tensors, highlight vertices that fail, return fracture plane for each fail
6: for each vertex to be duplicated do
7: Duplicate the vertex that fails
8: Assign each element to original or new vertex based on which side of the fracture plane it is found

This can all be brought together to form Algorithms 5 and 6 that represent the total process for simulating the deformation and fracture using the finite element method.

To be able to complete these calculations using the graphics card in Unity requires the use of Compute Shaders that are written in HLSL [Unity Technologies, 2018b]. To incorporate these shaders into a GameObject within Unity a new ComputeShader asset must be created. This creates a .compute file in which all operations such as calculating the jacobians, performing the row-vector multiplications and calculating the stress tensors are written in kernels. To interact with these kernels from the C# script requires defining structs in both the C# script and in the shader, these structs define the data for
a single element, node or matrix row depending on their use. Passing the information back and forth between the shader is done through the use of ComputeBuffers, an array of these data structs is set to the buffer before the buffer is set to the shader. With the data passed to the shader, its dispatch method can be called to run the specified kernel. After which the getData method of the buffer can be called to get the processed data back from the shader.

The processes outlined here should provide a strong start to implementing the core elements of the finite element method in Unity. This details the implementation of a simulation of a brittle material, omitting the small additions discussed in [O’Brien et al., 2002, Parker and O’Brien, 2009b] that enable the method to simulate ductile fracture. It can be seen that this method and its implementation are not trivial, and that the complexity of the tasks performed requires significant optimisations to achieve the strict performance goals of a real-time application.

3.5 Conclusion

This chapter has provided the necessary information to implement the methods used in this project, as well as how to repeat the experiments conducted. It gives detailed explanations of the approaches used to evaluate the pre-fractured and mass-spring methods.

It was decided to use the explicit Euler integration scheme for performing the mass-spring simulation, while this is the faster of the two integration schemes it is less reliable. In this project, it was decided to use the explicit scheme, due to its speed and that it is the preferred method in the literature. If it was deemed necessary to have the stability offered by the use of the implicit integration scheme in future work, the key elements described implementing the finite element method could be transferred to the mass-spring method. Specifically the use of the conjugate gradient analysis to solve the system of equations, the use of compressed sparse row lists to store the matrices that represent the system of equations, and finally, the process of breaking the algorithm down into chunks that are parallelisable, to take advantage of the performance capabilities of modern graphics cards.
Chapter 4

Results and Discussion

This chapter aims to present the results of the experiment and survey detailed in Chapter 3, to determine with quantitative evidence if there is any difference in the performance of the two methods tested. While the survey results aim to answer the question if there exists a perceived difference in the quality between the two methods. As the method is quite complex and its implementation is out of the scope of this project, the finite element method was only being explored in this project and there are no results associated.

4.1 Performance

The raw data recorded during the performance tests detailed in Chapter 3.3.1 was the time to render each frame. The mean times to render each frame across all 100 runs of the experiments was calculated and is shown in Figure 4.1, while the inverse of this, the mean instantaneous framerate of the experiments is shown in Figure 4.2, as it provides a measure of performance in commonly used units for the field. The recording starts when the objects are created in the scene and ends a short amount of time after the collisions happen. It can be seen that for both methods there is a performance hit when creating the object, this is especially significant for the mass-spring method, which not only has to read from the file to obtain the vertices, surface triangles and tetrahedral elements, it must search through the system to determine where to add springs and create each of the individual GameObjects required for the many tetrahedral elements.
Figure 4.1: Comparison of mean time to render each frame

Figure 4.2: Comparison of mean instantaneous framerate
While the pre-fractured method simply reads the file and creates a single mesh for the one GameObject.

With the pre-fractured model, once the initial hit has passed and the object has been created, the performance stays steadily at 50 FPS, that is until the collision occurs to fracture the object. Looking at both Figures 4.1 and 4.2, there is significant slowdown around the 0.8 s mark when the object is hit. At this point in the pre-fractured model, the GameObjects for each of the individual fragments are being created. In the frames, following there are many more collisions with fragments colliding with other fragments, and with the sphere fired at the object. Shortly, after the performance picks back up, as the fragments of the object start to settle on the floor of the scene.

It should be noted that the movement of the sphere through the scene is handled through Unity’s physics engine, which runs on its own separate thread. Meaning that while the creation of the mass-spring object may have brought the scene to a standstill of 3 FPS, as shown on Figure 4.2, the position of the sphere, is still being updated regularly. So with the positions of both the sphere and the destructible object the same, and the same force provided to the sphere in every experiment, the sphere will still collide with the destructible object at the same time no matter how many frames have been and passed since the start of the experiment.

For the mass-spring model, once past the significant reduction in performance in creating the object, its performance remains relatively stable throughout the collision and its aftermath. There is a slight dip around the collision at the time 0.8 s, however, this is not a significant change from the performance after when the fragments start to settle. For the mass-spring method, all the individual GameObjects have already been created before the collision preventing this from impacting performance when it is hit. With this method, the fracture is performed by removing Spring objects from the system, which is not an expensive task through the method’s use of Lists that allow for efficient removal of objects in the container. Instead, the hit to the performance comes from the increased number of physics calculations being performed on the many elements that make up the mass-spring object. As the individual tetrahedral elements always exist, even small external forces such as gravity pulling the object into the floor will result in the springs connected to push back resulting in many small motions constantly occurring within
the object. This results in a consistently lower framerate possible with the mass-spring method due to the higher number of physics calculations necessary.

While Figure 4.2 shows the mean values of the instantaneous framerate across time, Figure 4.3 shows the standard deviations for each of these mean values. It can be seen that the standard deviations for the pre-fractured method, fluctuate mostly between 7 to 8 FPS, while the standard deviations for the mass-spring method are slightly more consistent at 5 to 6 FPS. The only major inconsistency in the standard deviations comes from the fracture event of the pre-fractured model, at which point the standard deviation doubles in size, this could be evidence that the fracture event isn’t always happening at the same time in each experiment and is spread across a number of frames.

To complete the aim of the project to compare these methods for modelling fracture, it is necessary to perform significance testing to find if there is a difference in the performance between the two methods. The null hypothesis for this significance testing is attempting to prove false is that there is no difference between the performance of the two methods. Using the mean time to render each frame for both methods the t-test was performed. This showed that the null hypothesis was false and that there is a difference between the performances of the two methods, \( t(118) = -3.0692, p = 0.0027, r = 0.2719 \) with

![Figure 4.3: Standard deviations of framerate](image)
the change between the two methods having a medium effect size. The mass-spring is slower with a grand mean framerate of 33.2611 \textit{FPS} as compared to the 45.5941 \textit{FPS} of the pre-fractured method.

## 4.2 Survey

![Image of bar chart]

\textbf{Figure 4.4:} Pre-fractured: The destruction of this object looks realistic

The survey succeeded in obtaining 19 participants, all of which answered every question. The data obtained through the survey aimed to show if one method was perceived as more realistic than the other. As discussed in Chapter 3.3.3, this was achieved by asking participants to rate how much they agreed with the statement "The destruction of this object looks realistic" using a Likert scale. Figures 4.3 and 4.4 show the results of these Likert scales. The average score for the pre-fractured method is 4.3158, with a standard deviation of 1.6884 bringing it just above the "Neither agree or disagree" category. While the mass-spring method scored an average value of 6, with a standard deviation of 1.4907, cementing it in the agree category. As can be seen from the figures, the mass-spring method is, for the majority distributed within the agree category with a mode of 7 which is "Strongly agree", while the pre-fractured is more evenly distributed.
Figure 4.5: Mass-Spring: The destruction of this object looks realistic across the entire scale with a mode of 5. This shows that participants were more certain that the mass-spring method looks realistic, while the pre-fractured method is more divisive, only slightly tipping towards looking realistic.

As with the performance tests, it is necessary to observe if there exists a difference between the perceived realism of each method. This is to be achieved using significance testing specifically using the t-test. The null hypothesis for this t-test is that there is no difference between the perceived realism of the two methods. The results of the t-test were $t(18) = -7.6711, p = 3.7635 \times 10^{-7}, r = 0.8775$, showing that the null hypotheses is false and these results are significant with a large effect size.

To investigate the reasons why participants rated each method the way they did, the second question for each method asked "Why does the destruction look realistic? (Tick all that apply or none if unrealistic)". As discussed in Chapter 3.3.3, participants were given categories to choose from, as some found it difficult to figure out why a method looked realistic to them or not. As can be seen in comparing Figures 4.6, and 4.7 for both methods the movement of the fragments looked realistic, as expected with Unity’s physics engine, which handles the movement of all rigid bodies in the scene, so this result is as expected.
Chapter 4. Results & Discussion

Figure 4.6: Pre-fractured: Why does the destruction look realistic?

Figure 4.7: Mass-Spring: Why does the destruction look realistic?
The more interesting numbers to look at are for the "The object breaks into pieces as expected" and "The fracture of the object occurs where expected", as while both methods use a tetrahedral mesh, the mass-spring approach allows for local fracture of the object. This allows for the mass-spring method to break down into bigger more realistic chunks than straight into sharp tetrahedrons everytime like the pre-fractured method. For the pre-fractured method, both these criteria got a frequency of 8, roughly 42% of the sample, while for the mass-spring method it was 15 and 16 respectively, equating to approximately 79% and 84% of the sample. This shows that inclusion of the local fracture capable in the mass-spring method is noticeable, pairing this with the results of the previous two questions shows that it has a significant effect on the perceived realism of the method.

The penultimate question of the survey simply asked "Which of the two methods do you prefer the look of?", pre-fractured received 21% and mass-spring received 79% of the sample. The final question asked participants to explain their answer to the previous question. The majority of the responses were simply, the method chosen was more realistic than the other. However, 6 of the 19 responses specifically mentioned the local fracture of the mass-spring method as making the visual appearance more realistic, as their reason for preferring the mass-spring method.

4.3 Conclusion

This chapter has presented the results of the experiment and survey detailed in Chapter 3. It shows that the results of the performance tests are significant, with the mass-spring method proving to be consistently slower than the pre-fractured method. The mass-spring method suffered from a significant performance drop when the object is being generated due to the creation of a large number of GameObjects and searching through the system to connect the springs. While the pre-fractured method drops in performance slightly at the creation of the object but it is not until the fracture event that a major performance hit occurs, as there are many fragments being generated.

As discussed in Chapter 3, the mass-spring method could be sped up by moving the search to find which elements are connected by springs to a separate program. By moving this component to the pre-production, it would remove the long process of searching
through each unique pair of elements at runtime. However, as can be seen in Figure 4.2, the mass-spring method is consistently slower than the pre-fractured method apart from when the fracture occurs. Therefore, while moving the search to pre-production would improve the performance of the mass-spring method, it is likely that this would not change the results of the comparison.

The survey showed, however, that the mass-spring is perceived to be visually more realistic than the pre-fractured method. With data showing that this is related to the pieces it breaks into, and to where the fracture happens, two points that are linked to the local fracture that is capable through the method.

It should be noted, that the use of the explicit Euler scheme for the mass-spring limits how stiff the system can be. As discussed in Chapter 2.3.1.1 if the stiffness of the spring is too high for the timestep used, then the spring may deform further from the rest length in the other direction, leading to the system to explode. Through the implementation and testing of this method, it was discovered that for the ico mesh used in testing and with elements connected if they shared an edge, the max stiffness was roughly 140 N/m. Any higher and even the lightest touch to the object would cause large movements in the springs until they reached their critical length ratio, and sent pieces of the object flying. While this was expected behaviour, the low stiffness meant that the object itself was not very stiff, it suffered from large amounts of elastic deformation to stimuli. While this is perfectly acceptable for some materials, it does mean that the method is limited and cannot simulate tougher materials like stone which would have very little deformation before fracturing. If a developer wished to use the mass-spring method for a stiff material such as stone or steel in a real-time application that has large timesteps, then it is recommended to use an implicit integration scheme like that which is used in the finite element method discussed in Chapter 3.4.

This chapter completes the final two objectives of this project laid out in Chapter 1, to determine if there exists a difference in performance between the two methods and if there is a perceived improvement from the use of more computationally expensive methods.
Chapter 5

Conclusion

To conclude this project set out to, compare a number of methods for modelling fractures, for use in real-time applications. It did so through the implementation of a geometric-based method and a physically-based method, the pre-fractured method and the mass-spring respectively.

The performance of each of these methods was compared, by recording the time to render each frame throughout a fracture event. By repeating the experiment it was possible to prove with the use of the t-test, that there exists a difference in the performance, the reported results of the t-test are $t(118) = -3.0692$, $p = 0.0027$, $r = 0.2719$, with the mass-spring method shown to be slower with a grand mean framerate of 33.2611 FPS as compared to the pre-fractured method with a grand mean of 45.5941 FPS.

The second point of comparison between the two methods was to investigate if there existed a perceived improvement from one method to another. This was accomplished through the use of a semi-quantitative survey. That asked participants to rank how much they agreed with the statement ”The destruction of this object looks realistic?”, on a seven-point Likert scale. The results of this showed that the pre-fractured method averaged at 4.3158 corresponding to slightly above ”Neither agree or disagree” on the scale, while the mass-spring method proved to look more realistic to participants with a mean of 6. Once more the t-test was used to determine the significance of the results that there exists a difference in the perceived quality of the two methods, the results of which are $t(18) = -7.6711$, $p = 3.7635 \times 10^{-7}$, $r = 0.8775$. 

59
With these results, it can be seen that there does exist a perceived improvement in the quality of the destruction obtained through the more computationally expensive method, in this case. This result goes to show, that out of the methods evaluated neither method stands above, as the best in every scenario. Therefore, it is up to the developer to choose a method that creates an appropriate balance between the performance of their game and the realism of the virtual world they are trying to craft.

5.1 Future work

The project also looked to explore how to implement the finite element method, as this is a powerful approach to accurately modelling deformation and fracture. Unfortunately, the complexity of the method and the amount of optimisation it requires to be viable for real-time use meant that the implementation of this method was impractical, for the timeframe of this project. The goal of exploring the implementation of this model in Chapter 3.4, was to provide a detailed account of how the equations that build the model are linked, as well how the implementation could be achieved. In doing so, this research could be continued in the future, by implementing the finite element method, to allow for a more accurate comparison of it with other methods such as the two implemented in this project.

As discussed previously the use of the explicit Euler integration scheme was chosen due to its speed and that it is the preferred method in the literature. However, this scheme is not stable for stiff systems, which leaves the mass-spring method limited in possible uses. Possible future work could focus on the implementation of the mass-spring method, using the implicit integration scheme. Many of the details discussed in Chapter 3.4, for the implementation of the finite element method are transferable to the mass-spring method. Once implemented it would be critical to review if this advancement provided significant improvement to the method and to see how it compares to other methods such as the finite element method.

Finally, the methods explored and evaluated in this project were limited to the simulation of brittle materials. As discussed in Chapter 2, both physically-based models are capable of simulating more ductile materials through small additions to their systems. The question it raises is, does the addition of plastic deformation improve the quality of the
destruction enough to be worth any additional cost in computational time or in time to implement.
Appendix A

Finite Element Method
Supplemental Algorithms

The implementation of the finite element method refers to a number of processes or containers that may or may not be known to the reader. These are briefly described here to provide more complete instructions to the whole finite element method created by [Parker and O’Brien, 2009b].

A.1 Conjugate Gradient Method

The conjugate gradient method is a method of solving systems of equations iteratively. It takes a system of equations in the format of $Ax = b$, where $A$ is a known matrix, $x$ is a column vector that the solver is trying to find and $b$ is a column vector of known values. Conjugate gradient iteratively improves the value for $x$ by moving along different search directions until it converges on the solution. Equation 3.5, may not look it but is of the format $Ax = b$, the left-hand side is the matrix $A$, $v^+$ is the column vector $x$ and the entire right-hand side once calculated is a column vector. The process is outlined in Algorithm 7, where the subscripts refer to the iteration of the solver, $x_0$ refers to the initial values of the column vector $x$, while $d_i$ refers to the search direction of the current iteration and $d_{i+1}$ refers to the search direction of the next iteration.
Algorithm 7 Conjugate gradient method
1: Find the first search direction: \( d_0 = r_0 = b - Ax_0 \)
2: while relative error \( \geq \) acceptable OR iterations \( < \) max do
3: Find how far to move in this search direction: \( \alpha_i = \frac{r_i^T r_i}{d_i^T A d_i} = \frac{r_i \cdot r_i}{d_i \cdot A d_i} \)
4: Improve upon the value of \( x \): \( x_{i+1} = x_i + \alpha_i d_i \)
5: \( r_{i+1} = r_i = \alpha_i A d_i \)
6: \( \beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} = \frac{r_{i+1} \cdot r_{i+1}}{r_i \cdot r_i} \)
7: Find the next search direction: \( d_{i+1} = r_{i+1} + \beta_{i+1} d_i \)
A.2 Inverse of a Matrix

Algorithm 8 Inverse of a 3 × 3 matrix
1: if determinant not equal to 0 then
2:     for n = 0 to 2 do
3:         for m = 0 to 2 do
4:             ⊿ Get the sign for the value at position n,m with the following:
5:             sign = 1
6:             if n modulus 2 is not 0 then
7:                     sign = -1
8:             if m modulus 2 is not 0 then
9:                     sign = -1
10: Temporary matrix at position n,m equals sign times the determinant of the 2 × 2 matrix obtained by removing row n and column m from input matrix
11: Adjoint matrix equals the transpose of temporary matrix
12: Inverse equals the adjoint divided by the determinant of the input matrix

A.3 Polar Decomposition Algorithm

The polar decomposition, breaks the given matrix $F$ into a orthonormal component $Q$ and a symmetric component $A$. [Parker and O’Brien, 2009b] use this to factor out rotation from the strain and stress calculations. The process is shown in Algorithm 9

Algorithm 9 Polar decomposition
1: $D = v^{-1}Av$, find the diagonal matrix D, $v$ is the eigenvector matrix, found using QR algorithm
2: $\sqrt{F^T F} = vD^{\frac{1}{2}}v^{-1}$, find the square root of the values on the diagonal matrix
3: $A = \sqrt{F^T F}$, get the symmetric component
4: $Q = FA^{-1}$, get the orthonormal component
A.4 QR Decomposition and Algorithm

The QR algorithm provides a method of iteratively calculating the eigenvectors and eigenvalues of a given matrix using QR decomposition, the process is shown in Algorithms 10 and 11.

Algorithm 10 QR decomposition
1: Starting with matrix $A$, take each column as $a_1$, $a_2$, $a_3$ ... $a_N$
2: $u_1 = a_1$, $e_1 = \frac{u_1}{\|u_1\|}$
3: $u_2 = a_2 - (a_2 \cdot e_1)e_1$
4: $e_2 = \frac{u_2}{\|u_2\|}$
5: $u_3 = a_3 - (a_3 \cdot e_1)e_1 - (a_3 \cdot e_2)e_2$
6: $e_3 = \frac{u_3}{\|u_3\|}$
7: Repeat for each column, each time performing dot product and scalar vector multiplication with each previous $e$
8: Matrix $Q$ is constructed from each column vector $e$
9: The upper triangular matrix $R$ is constructed of the dot products of each $a$ and $e$ such that the index of $e$ is the row and the index of $a$ is the column, the lower triangular elements are all zero

Algorithm 11 QR Algorithm
1: Start with $A_0 = A$
2: $v = I$, eigenvectors, start as identity matrix
3: while Solution has yet to converge do
4: $A_{k-1} = Q_kR_k$, perform QR decomposition of previous $A$
5: $A_k = R_kQ_k$, Find the new $A$ from the decomposition
6: $v = vQ$, multiply previous eigenvector matrix by $Q$, to improve upon eigenvectors
7: The values along the diagonals of $A$ are the eigenvalues, while the columns of $v$ are the corresponding eigenvectors

A.5 Compressed Sparse Row List

The compressed sparse row list is a container, that allows for efficient storage of sparse matrices, while also allowing for fast access for row-vector calculations. It consists of three arrays $A$, $IA$, and $JA$. $A$ contains all of the non-zero values of the sparse matrix in row-major order, $JA$ stores all of the corresponding column indices for each non-zero value in $A$ and $IA$ stores the indices for the array $A$ of the first non-zero value in each row. $IA$ is a little more complex, while the previous two arrays are of the length of
the number of non-zero values, $IA$ is of the length $\text{rows} + 1$, it is defined recursively as $IA_i = IA_{i-1} + \text{number of non-zero values on } i^{th} \text{ row of the original matrix.}$

To access the values within this container for any row, for example when computing row-vector multiplications, simply loop through the values of $A$ between the indices $IA_i$ and $IA_{i+1} - 1$ as all the values of $IA$ represent the start of each row in the array $A$. To retrieve the column indices for each of these non-zero values is done the same way through looping through the values of $JA$ between $IA_i$ and $IA_{i+1} - 1$.

### A.6 Coordinate List

The coordinate list, is a simple container of three arrays that allow for more efficient storage of sparse matrices [Pissanetzky, 1984], the first an array of all the non-zero values of the matrix, the second an array of all of the row indices for the corresponding non-zero values and finally an array of all of the corresponding column indices for the non-zero values of the matrix.
Bibliography


