A Cracking Algorithm for Destructible 3D Objects

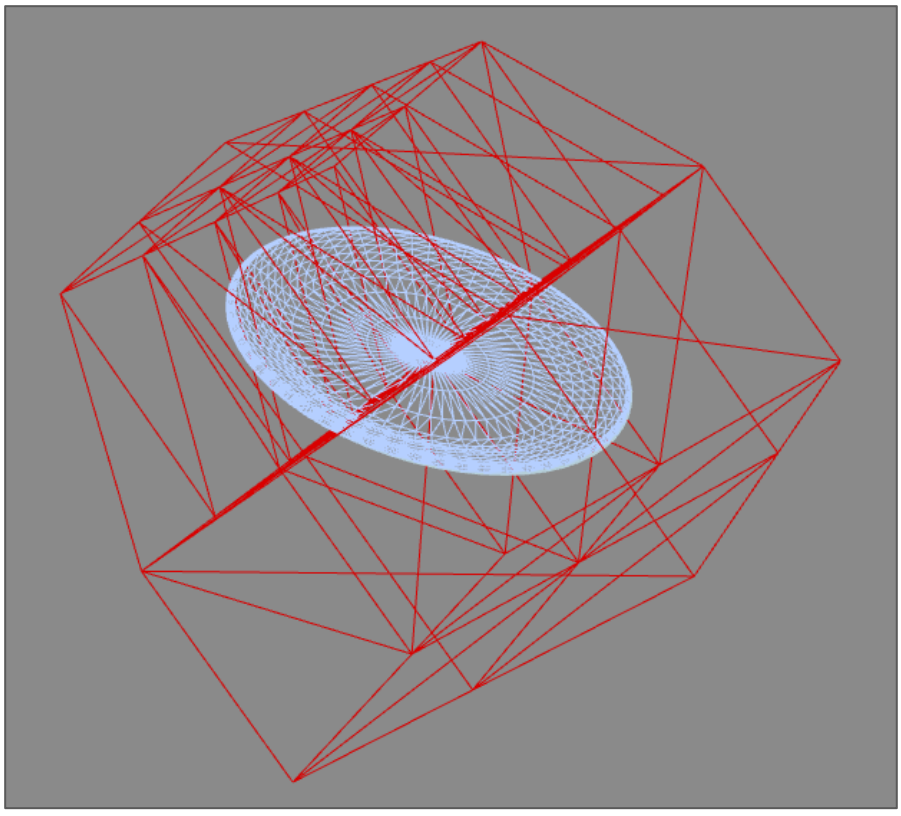
By Steven Workman

Supervisor: Dr. Steve Maddock

Module Code: COM3021

3rd May 2006

This report is submitted in partial fulfillment of the requirement for the degree of Master of Science in Computer Science by Steven Workman



Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Steven Workman

Signature:

Date: 3rd May 2006

Abstract

In modern computer games, realism in the environments is proving a major factor in the quality of the game. As computers become more powerful, these environments can be created in immense detail. Recently, realistic interaction with objects in the environment has proved to be very useful for realism and for the game play experience. One area of this interaction is the destruction of objects in the game world. This area has not been properly explored in computer games and truly destructible environments have never been created.

The aim of this project is to create an algorithm for the real time destruction of three dimensional objects. This algorithm will be able to be applied to any 3D mesh. A fracturing algorithm will be employed to realistically crack the object upon impact determined by a weakness alpha texture.

The visual results of this system are good but not without fault. The performance of the system is extremely encouraging and with a limit on the number of polygons to be cracked the technique can be considered to run in real-time.

Acknowledgements

I would like to thank Steve Maddock for his continued support throughout the past year and for motivating me to produce the best project I could. I would also like to thank:

- Tom Miller at Microsoft for producing Managed DirectX without which this project would not have been as successful
- Chris Yorkston and Alistair Jones for their input in group meetings whilst pretending to know what I was on about
- The DirectX community on GameDev.net who continued to answer all of my technical needs.
- Ben Clare for letting me break some of his plates “in the name of science”
- Finally to the Sheffield University Hockey Club, for many a welcome distraction every Wednesday evening.

Contents

List of Figures

List of Equations

List of Tables

Chapter 1: Introduction

This project is concerned with the realistic simulation of destructible objects. By creating a crack pattern and applying it to a 3D object, the object can split or shatter dependent on the amount of force applied to the object. The end result will be a crack generation program and a small testing program to demonstrate the 3D cracking process.

Realistic simulation of real-world events has become an important part of some of today's modern computer games. With the computer gaming industry growing further each year, more people are being exposed to computer gaming. In accordance with Moore's law [1] computers are doubling their processing power every 18 months. Since the advent of real-time 3D graphics in the popular game "Quake" (1995) by Id Software, gamers have demanded more realistic representations of environments to interact with. True photo-realistic rendering as seen in feature films such as Toy Story and Monsters Inc. is not currently possible in real time despite the ever increasing processing power available to developers.

Traditionally, destructible environments have been created by already having the object split in its post-impact state and animating these parts in a set way. This can lead to some very unrealistic effects such as demonstrated in the popular game "Half-Life 2" (2004) by Valve where striking a plank of wood at its extremities breaks the wood at the centre. Nor does it matter how much force has been applied to these so called destructible objects as they will deform in the same way if hit with a bullet or a grenade.

Similar work has been performed in previous years by Fox [2] and Miller [3]. Fox achieved this 3D cracking effect by generating a 2D pattern and extruding it in to 3D. Miller created his crack pattern by joining layers of 2D crack patterns using binary space partitioning. Their work will be discussed in greater detail in the literature analysis section of this report.

This work intends to create two programs. One program creates a crack pattern guided by a strength texture and forms a 3D polygonal object from this pattern. This object is then used in the second program which performs the cracking of the target object. The former program is an 'offline' program and the latter is a real-time system.

Chapter 2 of this report contains a review of literature relative to this project including fracture mechanics, methods of fracture simulation, texturing, polygon clipping and animation.

Chapter 3 is the requirements and analysis section. This area details the creation of a crack pattern, how that pattern is affected by an alpha texture, the application of that pattern to an arbitrary object, intersection methods and the cracking process, the user-

interface and requirements of the system. It also describes how the project will be evaluated and tested.

Chapter 4 contains the details of the design of the system including the decision to separate the two systems in to an offline and online process. It also discusses the major methods for performing the task and chooses the best method to implement.

Chapter 5 documents the implementation of the two systems. It contains a detailed breakdown of the path-finding mechanism in the fracture generator and the cracking mechanism in the fracture viewer. It also documents any problems found during the evaluation.

Chapter 6 contains the evaluation of the project. It contains comparisons to real-world tests and also includes performance charts for the cracking process. This section also includes an analysis of any problems or errors in the system and suggestions for future work in this area.

Chapter 7 concludes the project and determines its success.

Chapter 2: Literature Review

Section 2.1 investigates fracture mechanics in brittle materials; section 2.2 discusses offline and real-time methods of implementing 3D fracturing; section 2.3 covers the application of a crack pattern to an object via a texture. Sections 2.4 to 2.6 investigate non-innovative parts of this project such as polygon clipping and animation.

2.1: Fracture Mechanics

This section contains an overview of fracture mechanics. The relevant reading for this can be found in [6]

2.1.1: Material Properties

All materials in the world are made up of molecules. These molecules are held together by inter-molecular bonds, the amount of force between the molecules depends on the type of inter-molecular bond and the state of the material (solid, liquid or gas). In solids, molecules are arranged in a lattice formation with ionic and van der Waals bonding holding the molecules together, as shown in figure 2.1.

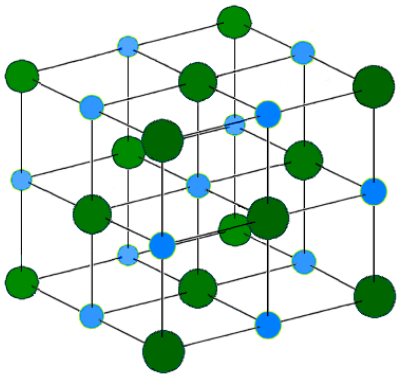


Figure 2.1 - Inter-molecular bonding in Sodium Chloride (NACl). Taken from [4]

When a force is applied to a solid, the bonds between the molecules experience stress, a compressive action, or strain, a tensile action. These terms can be mathematically defined as:

$$\text{Stress} = \text{Force} / \text{Area}$$

$$(\text{Units: Nm-2} = \text{N} / \text{m2})$$

Equation 2.1 – Stress

$$\text{Strain} = \text{Extension} / \text{Original Length}$$

$$(\text{Units: none})$$

Equation 2.2 – Strain

When stress or strain is applied to a solid it undergoes one of two types of deformation:

- Elastic Deformation – This type of deformation is reversible. That is, once the force is no longer applied the object will return to its original shape. The relationship between the size of the deformation and the load applied to the object is called Hooke's Law. If an object exceeds its "elastic limit" then the object will become permanently deformed.

- Plastic Deformation – Once an object has exceeded its elastic limit, plastic deformation occurs. The linear relationship between load and deformation length no longer applies. Once the force has been removed then the object will not return to its original state. The additional region gained by plastic deformation is called the "Plastic Region".

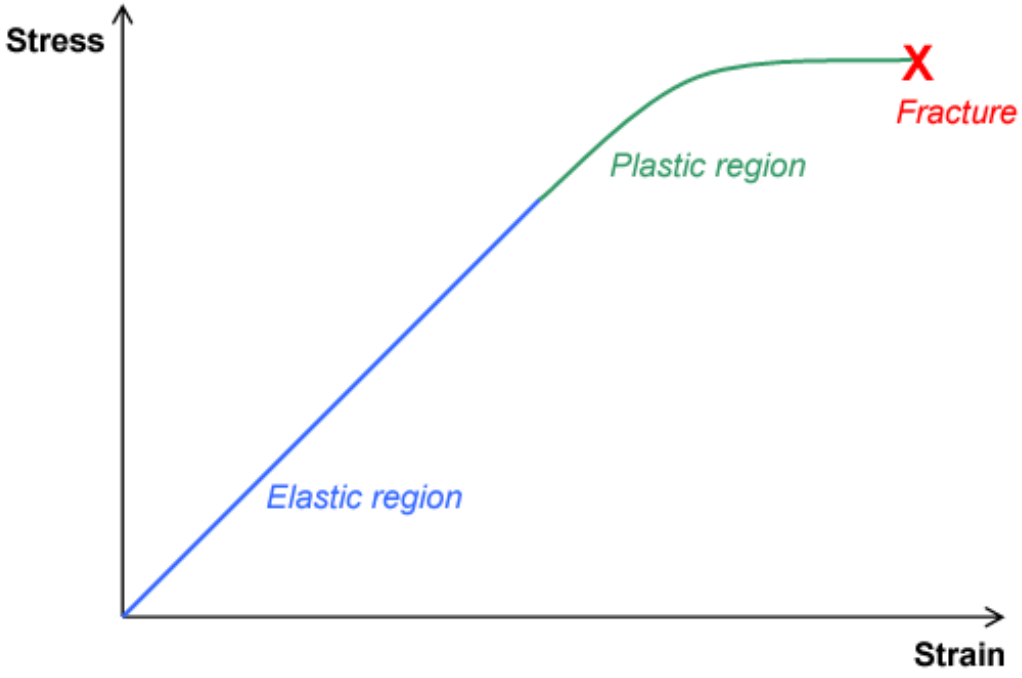


Figure 2.2 - A Stress/Strain graph from [5]

A break occurs after the material has reached the end of the elastic, and then plastic, deformation ranges. At this point forces accumulate until they are sufficient to cause a fracture. All materials will eventually fracture, if sufficient forces are applied.

2.1.2: Material Fracture

Fracture occurs in one of two basic ways, brittle fracture and ductile fracture. Ductile fracture takes place after extensive plastic deformation. This project is not concerned with modeling plastic deformation and so only brittle fracture will be considered.

2.1.3: Fracture in Brittle Materials

A brittle material is considered one that requires a very high load to initiate fracture (a strong material), but once the fracture has begun it propagates quickly as the material absorbs very little fracture energy (as brittle materials have a low toughness).

In brittle fracture (sometimes known as 'fast fracture'), no plastic deformation occurs before fracture. A crack is formed in the material because the stress applied to the inter-molecular bond is greater than the strength of the bond. This causes the inter-molecular bond to break. The molecules then separate and apply strain to the inter-molecular bonds next to them. This is called crack propagation.

[6] States that, "…crack extension (fracture) occurs when the energy available for crack growth is sufficient to overcome the resistance of the material."
The path of a fracture through a structure does not have to be along a single path or in a perfectly straight line. Crack branching is the effect caused by some of the intermolecular bonds, lying to the side of the main fracture path, absorbing some of the energy from the main fracture. Crack meandering demonstrates the principle that, from [7], "Cracks choose the path of least resistance", and therefore the crack tip will propagate through the bond which requires the least amount of energy to break.

2.2: Crack Simulation Research

Many attempts have been made to simulate fast fracture in a 3D model. These methods are either offline (not able to be performed in real-time) or in real-time. This project is concerned with real-time approaches; however, many of the techniques used in offline crack simulation can be adapted to suit real-time applications.

2.2.1: Offline Methods

Offline methods have many advantages over real-time methods. Since there is no constraint on rendering time, any calculations can be performed to their full accuracy in order to achieve the most realistic result. The target of the fracture method does not have to be a regular 3D polygon mesh. It may have special properties such as inner vertices or additional properties for each vertex in the mesh.

Work on offline methods for fracture simulation began in 1988 with Terzopoulos and Fleischer's papers [8, 9] into modeling deformation. Their method used energy functions to provide the basis of the continuous deformation model, demonstrated by modeling sheets of paper and cloth tearing.

O'Brien & Hodgins [10] used a finite element model to simulate brittle fracture. A finite element method "partitions the domain of the material into distinct sub-domains, or elements as shown in figure 2.3"

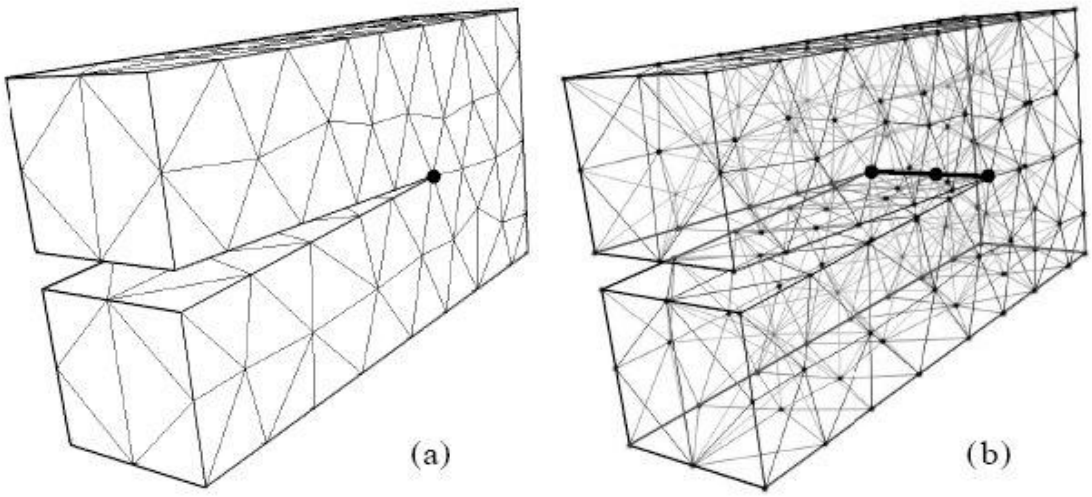


Figure 2.3 - Visualization of the Finite Element Method. Objects composed entirely of tetrahedrals. (Taken from [10])

The common vertices of this framework are known as 'nodes'. The stress placed on an object is modeled by considering the individual stress forces at the nodes connecting the finite elements. The strain is modeled by calculation of the distance of the nodes from their original positions in the framework.

This method is used to excellent effect to produce realistic deformation and destruction of the target object as shown in figure 2.4. However, this simulation takes approximately 75 minutes to render 1 second of simulation and unlike modern 3D computer games; the finite element model has an internal structure whereas modern 3D games tend to have hollow models. These hollow polygon meshes could be converted in to meshes compatible with the finite element model using a tool called NETGEN [11] but this process would be computationally expensive.

Hirota *et al* created their crack pattern model in [12] and [13] using a spring-network model. "In the spring-network model, the elements are represented by nodes. The connections between neighboring elements are represented by springs, and the connections between elements and the interior layers are shear springs" In this model, the stress forces applied to the nodes produce strain forces; when these forces breach the elastic limit of the springs connecting the node, a crack forms. The advantage of this method is that springs can extend without breaking and thus absorb some of the force applied whilst the crack propagates, therefore simulating a more realistic crack pattern.

Performance figures from [13] show that this method is highly expensive in computational terms as it takes 8 hours to render 25 hours of drying process for a cube model and 22 hours to render 55 hours of the process for a cantilever model. There are no figures demonstrating more complex object fracture.

Summary
Offline implementations are highly accurate and produce excellent results; however, they require specialist models or take a very long time to render. These methods teach us that computational accuracy has to be sacrificed in order to create real-time fracture models.

2.2.2: Real-Time Methods

A real-time method is one that a computer can compute and render to the screen 15 times per second or more. The rate at which a simulation takes place is called the frame rate and is measured in frames per second (fps). Rendering at a rate less than 15fps causes the simulation to either discard frames or slow down so that the simulation becomes jerky. With modern computer hardware, an ideal frame rate of 60fps (the point where the eye no longer notices jerking of the simulation) can be achieved for very complex scenes.

In order to achieve this frame rate, the accuracy of a simulation has to be sacrificed. General physical principles are maintained yet not to the same degree. In [12], Müller departs from the finite element model to a continuous one with excellent results. However, the simulation ran at a low frame rate on a computer not designed for home use and much more powerful than regular computers available at the time.

In current computer games, destruction and deformation effects are created by having the destructible object formed from the pieces it would be broken into. The destruction of the object is then triggered by an event in the game world. This produces excellent results and can be controlled exactly so that the desired effect can be achieved every time. This, however, is only available for objects that have been specifically designed for destruction and it can take a lot of time for artists and modelers to create these objects.

There is one exception to this, however; the popular game Red Faction (2001) by Volition used a technology called Geo-Mod (for an FAQ see [13]). This allowed players to destroy almost any surface, creating their own path through the game world. This approach was clever and not particularly costly in terms of performance. However, the realism of such explosions was not particularly high (as shown in figure 2.4), the effect could really only be applied to static objects and the resultant pieces of the object were not created and animated away from the explosion. Geo-Mod technology was also used in 2002's sequel Red Faction 2 but it has not been used since.

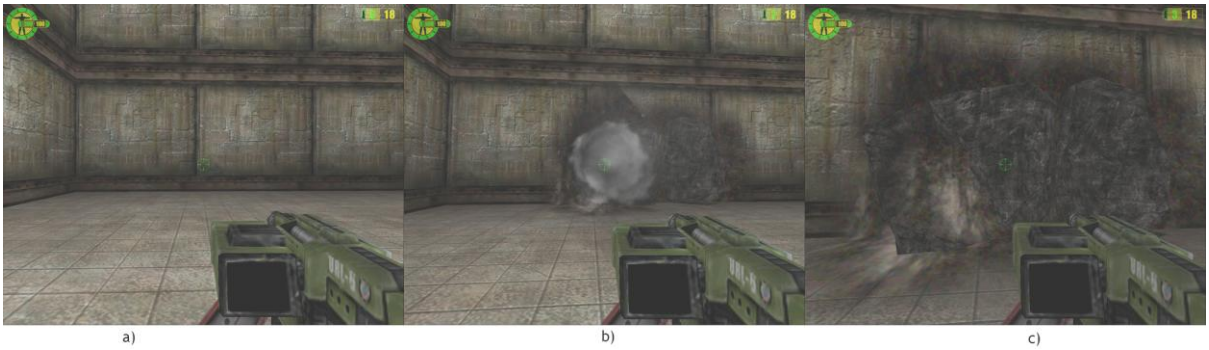


Figure 2.4 - Geo-Mod technology demonstration. a) shows an undamaged wall; b) shows the particle explosion effect covering the wall re-meshing; c) shows the deformed wall.

Fox's work [2] created a crack pattern to be applied to an object which would then be split based upon that pattern. Fox's pattern was created as a 3D texture by creating a 2D representation and then extruding the pattern in to 3D. The results of his work were visually very good however the shards of the resulting object were uniform in their depth, they were also unable to be clipped and their animation was linear after the explosion.

Miller's work [3] followed on from Fox in creating a similar crack pattern. However, Miller's work involved creating less detailed crack patterns inside the crack volume and joining these together producing more detailed shards. However, this produced problems for him and he had to use binary space partitioning (BSP) to solve his problems. The visual results, however, were very good.

Summary
Whilst Geo-Mod technology was successful for Red Faction, it does not produce a realistic crack in an object. Fox and Miller's work have been successful but their final techniques have either lacked three-dimensional accuracy or have produced artefacts in the final re-meshed models.

2.3: Applying a Crack Pattern

Once an acceptable crack pattern has been generated, it has to be applied to a 3D object. This shall be done by texture mapping. Watt's book [14] provides an excellent overview of texture mapping in 2D and 3D, however, I shall go over these techniques here.

2.3.1: 2D Texture Mapping

One of the reasons for the popularity of 2D-texture mapping is because of its application to the most common representation of three-dimensional objects – the polygon mesh. Watt [14] describes it as "a device that could be used to enhance the visual interest of a scene, rather than its photo-realism and its main attraction was its cheapness." What

this means is that a texture can be applied to a surface to simulate a higher level of detail without having to draw additional polygons. Figure 2.5 shows how textures can be applied to an object.

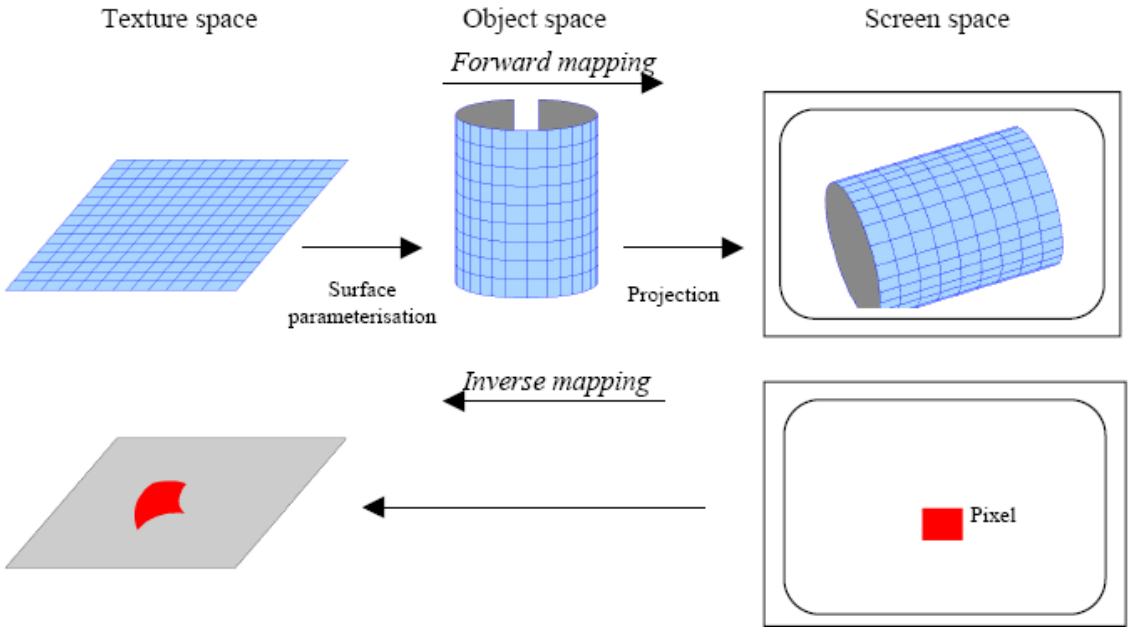


Figure 2.5 - Two commonly used methods of texture mapping, adapted from [14]

There are two ways to implement two-dimensional texture mapping, forward mapping and inverse mapping. Inverse mapping is where a pixels' colour (value) is determined by applying an algorithm to find out where on the two-dimensional texture the corresponding co-ordinates are. Forward mapping involves distorting the original texture onto the surface of the object, after which usual object space to screen space transforms can be used.

One of the drawbacks of 2D texture mapping is the additional requirement of texture co-ordinates (or UV co-ordinates) which are required to map the texture to a specific 3D object. This causes problems as the crack patterns that will be generated are designed so that any 3D object can be affected by them.

2.3.2: 3D Texture Mapping

A 3D texture map circumvents the problems of 2D texture maps by adding a third co-ordinate (W) to each texture map. This texture can then be considered as a volume. From [14] "Assigning an object a texture just involves evaluating a three-dimensional texture function at the surface points of the object". In general, 3D textures are procedurally generated due to the large memory requirements of 3D textures (n^3 bytes instead of n^2 bytes). However, this method solves these problems of object generalisation as any 3D object can have a 3D texture mapped to it.

2.3.3: Mipmapping

From [17], "The technique of mipmapping involves preprocessing a texture to create multiple copies, where each successive copy is one-half the size of the prior copy." Mipmapping is used in computer games to help manage the textures displayed on screen. Mipmaps, in conjunction with level-of-detail meshes, can reduce the memory footprint and bandwidth demands of textures.
In the context of this project, a mipmap is generated from an alpha texture to create a series of lower-resolution textures.

Summary
Texturing an object can provide additional detail to an object whilst using a highly reduced polygon count. However, 2D texture mapping is unsuitable for this project. 3D texture mapping requires more resources both in terms of memory and CPU time to apply the texture. However, it solves the problem of generalisation for the crack pattern.

Once the crack pattern has been applied to the mesh, it is possible to split the mesh in to fragments using these textured lines as a guide.

2.4: Polygon Clipping

Clipping polygons is an essential part of any 3D graphics engine. Rendering polygons that the user cannot see is a wasteful process and should be avoided. Clipping is a process that prevents any objects that are not represented by the user's view from being rendered. In modern graphics cards this function is now built in to the graphics pipeline.

[14] Presents the Sutherland-Hodgman clipping algorithm that clips a polygon to any given rectangle by testing each polygon edge against a single infinite clip boundary as shown in figure 2.6. This reduces the number of polygons being drawn on the screen at any one time and therefore increases the performance of the application.

Clipping is especially important in the context of this project as during the cracking process it is inevitable that the crack will cross polygon boundaries. Here, polygon clipping will be implemented to reconstruct the new mesh's edges. It is important to do this as otherwise the fragments could produce polygons which do not properly match the ones next to them leading to holes in the mesh.

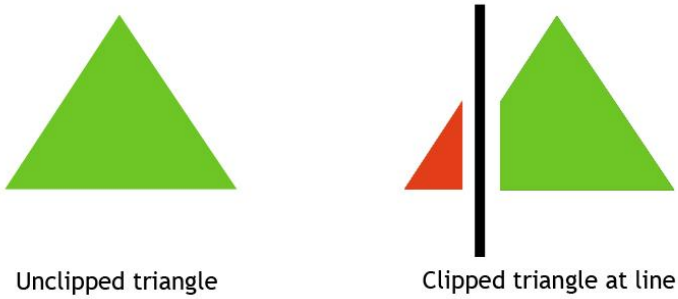


Figure 2.6 - Clipping

2.5: Animation in 3D graphics

The animation of an object in three-dimensional space is one of the fundamental tasks of many 3D programs. The ability to view a representation of a 3D object on a 2D screen would be a somewhat pointless task if there was no way to view the object from any angle. It is important to note that the basis for animation in 3D comes from work in 2D for example, Disney cartoons. This process sees individual frames shown one after another so that it looks like motion has occurred. The concept is the same in computer graphics; an object is transformed through space in increments per frame so that it seems that motion has occurred. In [14], Watt provides an excellent overview of this.

This project will use the explicit scripting technique outlined in [14] to animate the motion of the object fragments. The path that these shards will follow will be determined by real-world laws of motion, or kinematics. Details on Kinematics can be found in Appendix B.

The animation of the object's fragments is an important part of this project where the realism of the simulation is concerned. However, this project's main focus is on the crack pattern and the object fracture so this part of the project will be implemented last as it is not essential for the project to be successful.

Chapter 3: Requirements and Analysis

This chapter looks at the project in more detail. It analyses the method of creating a crack pattern and the application of that pattern. Section 3.1 shows an overview of the stages of the project. Section 3.2 details the creation of the crack pattern and how it differs from [2] and [3]. Section 3.3 discusses the alpha texture for the crack pattern. Section 3.4 analyses the application of the crack pattern to a 3D object. Section 3.5 covers the cracking process and re-meshing of the 3D object. Section 3.6 details the user interface of the programs; Section 3.7 covers the optimisation of the program and section 3.8 details the final requirements of the system.

3.1: Requirements Overview

This project is focused on the plausibility of the fracturing process. If the project is successful, a casual observer would be able to say that the simulation produces results that could realistically have come from the target 3D model. In the real-world, an object never fractures the same way twice; for this, a number of crack patterns should be created and at application time, the cracking algorithm can pick the most suitable crack pattern. This process must happen in real-time which, as discussed in section 2.2.2, means maintaining a frame-rate of over 60fps. The resultant pieces from the cracking process must then be animated away from the centre of the model in a realistic manner. This will add to the plausibility of the system. Figure 3.1 contains a flow diagram for these stages.

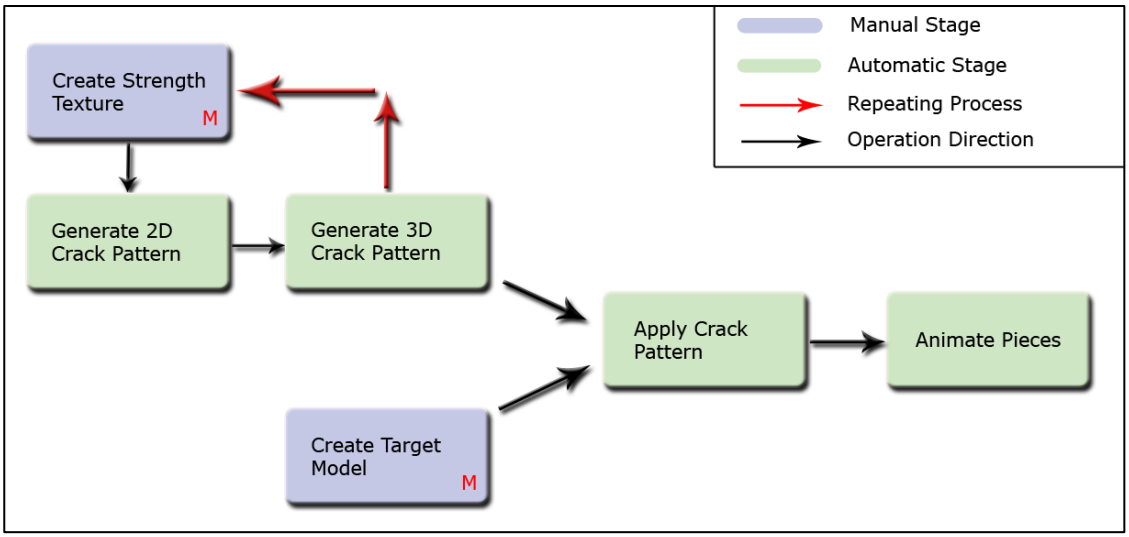


Figure 3.1 - Flow diagram of the system. A red M indicates a manual stage

3.2: Crack Pattern Creation

Creation of the crack pattern is possibly the most important task that this project will look at. It is the most labour intensive section and create the correct patterns requires the right tools for the job. Throughout this chapter, comparisons will be drawn between [2] and [3] as these are the works that are most relevant to this project. Their methods will be analyzed in-depth and their problems corrected by this project.

3.2.1: Crack Pattern Analysis

This section looks at ways of creating a realistic crack pattern from a simple algorithm.

Firstly, it should be stated that crack patterns for [2], [3] and this project are generated in a separate program offline and at runtime, the crack texture is applied to the object and then split. If the crack pattern was generated in real-time, the algorithm would have to be significantly simpler in order to create realistic effects without causing large performance drops.

In [2], Fox tries 2 methods of creating his crack pattern. The first way is using a block method where the crack propagates from node to node at ninety degree angles in a non-uniform manner. The second method that he tried was a more radial approach to the problem involving vectors. Fox had noted that cracks propagate in a “spider’s web” motion and this produced a more accurate visual representation as shown in figure 3.2.

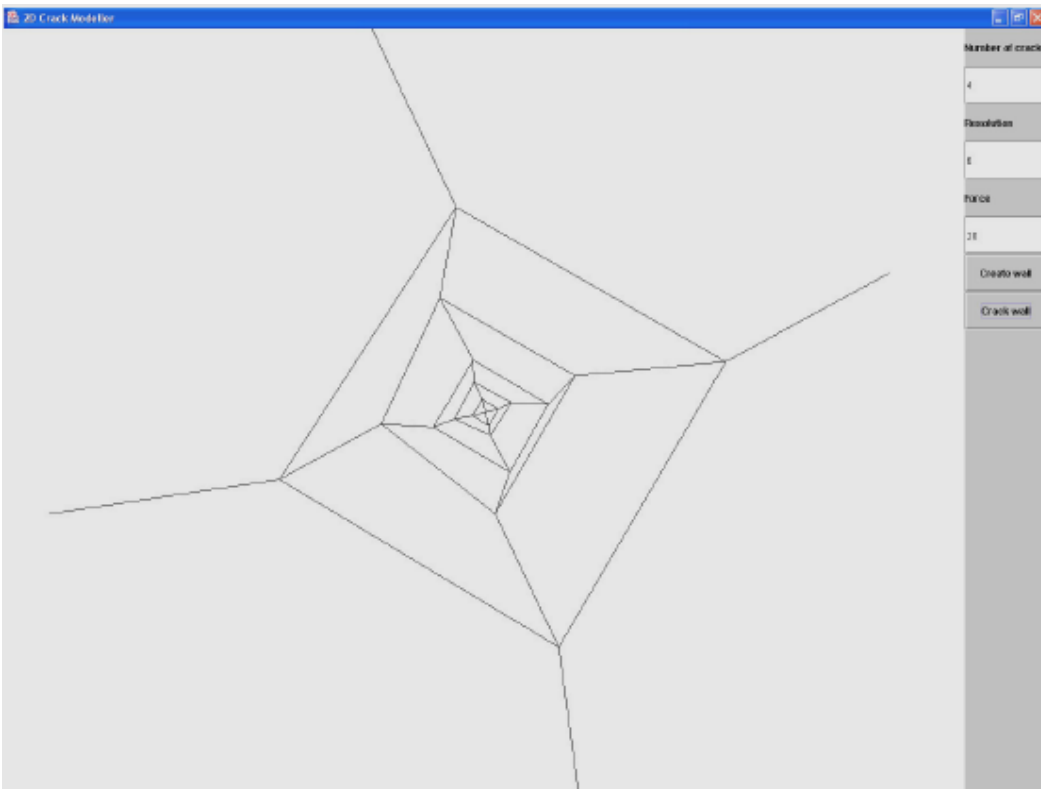


Figure 3.2 - 2D crack generation taken from [2]

In [3], Miller considers both of the ideas suggested in [2] and agrees with Fox that although a vector approach to crack pattern generation will make re-meshing the object difficult it produces a better crack pattern visually. Miller, however, disagrees with Fox's approach to creating the 3D crack pattern (a simple extrude of the 2D as stated in section 2.2.2) and states from [7] "In brittle fracture, the cracks run close to perpendicular to the applied stress." i.e. that there are many layers of crack patterns that are similar to each other that should be joined together to form a single 3D pattern.

Summary

It is the opinion of the author that, with hindsight, Miller [3] was incorrect to attempt a multiple-layer crack pattern using a vector approach. This approach caused him great difficulty because joining the multiple layers into acceptable shards when the cracks propagated through a floating point space meant that he had to use binary space partitioning (a good overview of this can be found in either [3] or [14]) to determine which crack line was where. It is believed that the multiple-layer approach to creating a crack-pattern is the correct one; however, in order to achieve simple construction of a 3D crack pattern, a discrete method must be used.

3.2.2: Crack Pattern Generation

From the research into brittle fracture, it was determined that a crack takes the path that requires least force to continue down. Since a discrete pattern is being used to describe the path of the cracks, it is appropriate that each of the nodes on a path be given a value of strength so that the crack may choose the weakest path travel on. This strength value could be given by a random number generator inside the program; however, this could produce wild patterns that in no way resemble a crack pattern. It was therefore decided that the strength of the crack pattern could be generated by an alpha texture (see section 3.3). This way, an artist can control areas of strength and weakness in an object i.e. if an object was marble surrounded by steel, the alpha texture would be weak in the centre and very strong around it. The alpha texture will be explored in more detail in section 3.3.

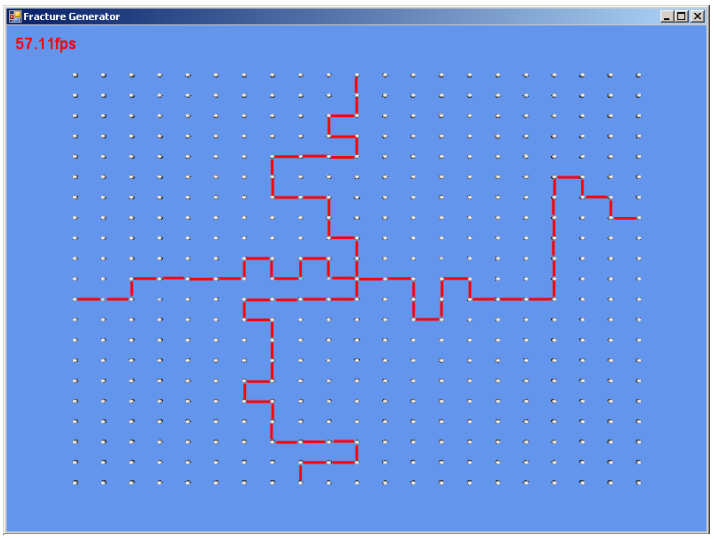


Figure 3.3 - A 4-star crack pattern

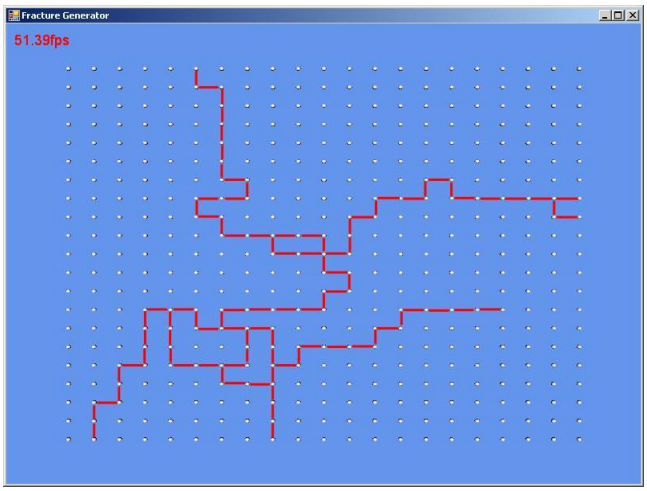


Figure 3.4 - A crack pattern with branching

Initial experiments with a crack pattern involved a simple star-like pattern with four crack propagating from a central point (figure 3.3). A crack like this had a number of properties. Each crack had a general direction (i.e. Up, Down) which prevented it from doubling back on itself. It also had a certain amount of force which decreased every node that it reached. From figure 3.3, it is clear that this simple crack pattern is not visually acceptable.

The research in to crack patterns showed branching whenever an area was particularly weak or a bond of similar strength was nearby. Initial experiments in to this branching (without nodes having differing strengths) produced better results (as shown in figure 3.4).

Whilst this did produce better results, it still did not match the visual quality of either [2] or [3]. It was concluded that one of the reasons for this was that cracks were not being given enough information to create a plausible crack pattern. The production of a good pattern is based on luck instead of a good algorithm. It is at this point that a strength value is required at every node so that the results of the fracture generator are reproducible.

3.3: Alpha Textures with a Crack Pattern

An alpha texture is a 2D texture with greyscale colouring that indicates transparency in an object. A 3D graphics engine reads a high white value (from a range of 0-255 where 0 is black and 255 is white) as a transparent surface and a low white value as being a solid surface with in-between values having varying levels of opacity. Transparency is the standard use of alpha textures; however, this project will be using them to convey the strength of certain areas of an object.

Figure 3.5 shows a very simple alpha texture. Looking at this texture it can be seen that the area in the top-right of the image features a higher white value indicating a weaker section of the object. Whenever a crack is propagating through this section it loses less force and performs a greater amount of branching as there is more energy to dissipate. The top-left section has a very low white value and it is unlikely that a crack will enter this area as it will choose a node with a larger white value such as the bright white strip travelling from the centre to the left of the image. As you can see from figure 3.6, the description provided above matches the crack pattern generated. This shows the star pattern branching towards the weak areas of the object. This, however, is not a very good example of an alpha texture as the gaps between the white pixels in the top right are entirely black. This means that if a node picks the black pixel as its colour then the crack will go 'off course'. One solution for this is to draw the fracture pattern at a higher resolution; another solution would be to create a better alpha *texture and a final solution would* be to allow the crack pattern to perform a heuristic search upon the nearby nodes to calculate the weakest path to follow. Figure 3.7 shows a high resolution crack pattern (10,201 nodes) with a radial alpha texture pattern.

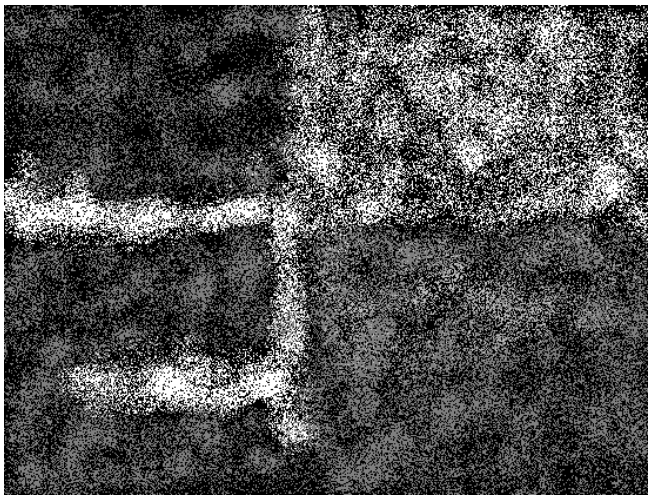


Figure 3.5 - A simple alpha texture

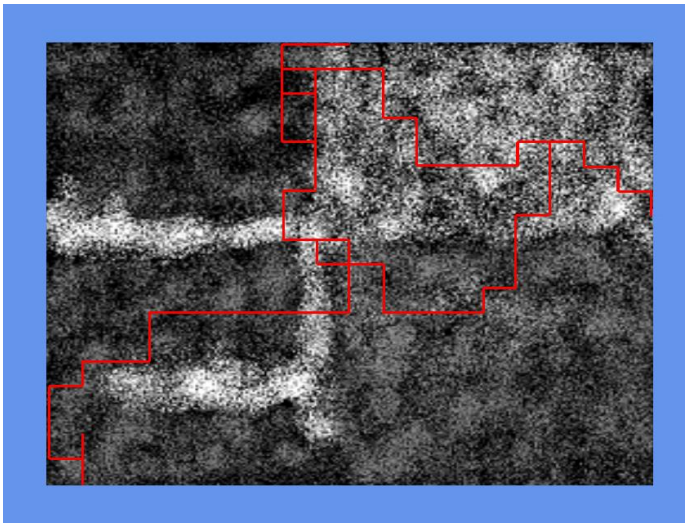


Figure 3.6 - A low resolution crack pattern following the alpha texture in figure 3.5

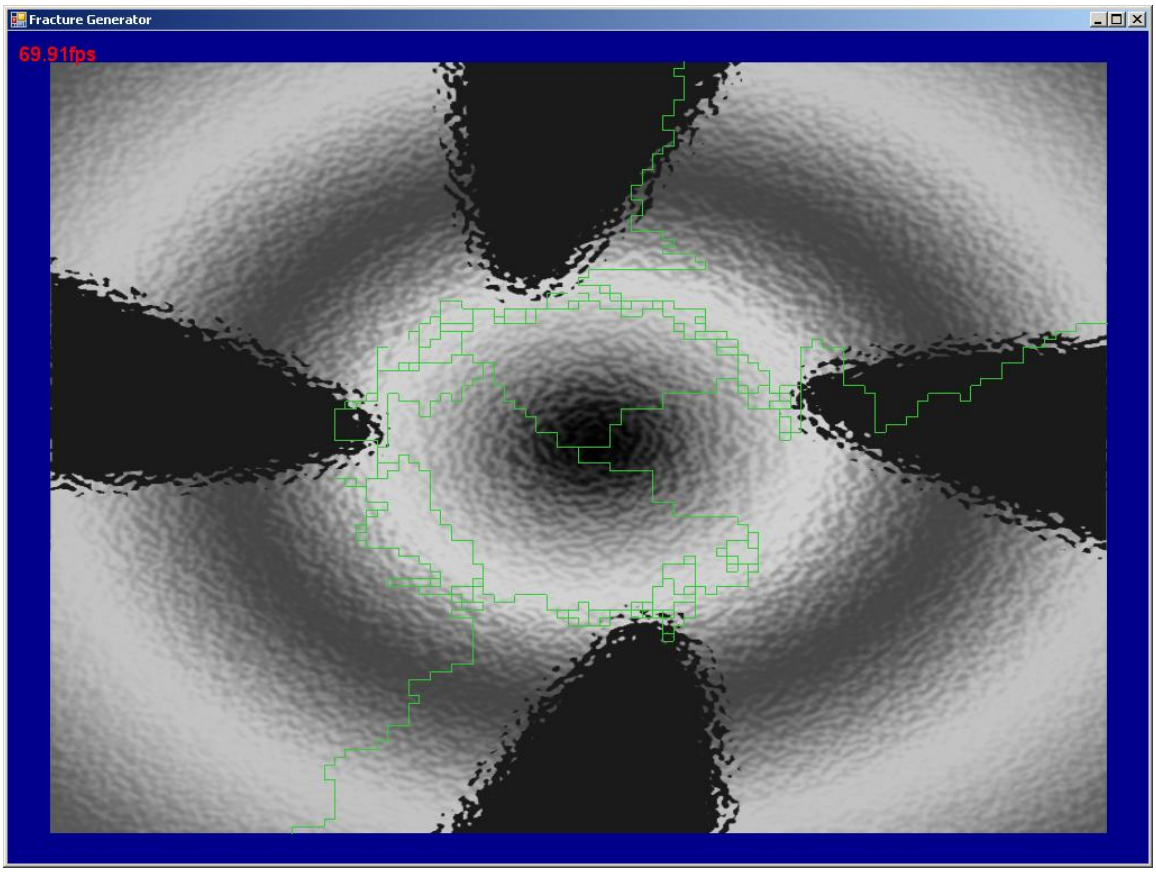


Figure 3.7 - A high resolution crack pattern

Figure 3.7 demonstrates a radial crack pattern created using an alpha texture. This result is promising, although it is clear that the crack pattern algorithm is not currently producing good results. The pattern in figure 3.7 would produce some excellent large pieces that would be visually plausible, yet the area around the radial crack appears to have shattered too much and would produce areas of 3D object that would either be too small to re-mesh effectively or simply have very abnormal shapes.

It should also be noted that whilst jagged edges are still present in the crack pattern lines, it is under consideration that in a later version of the project these 'jaggies' will not appear. Solutions under consideration at this time include diagonal lines between nodes, anti-aliasing and smoothing of the resulting meshes.

3.3.1: Creating Multiple Crack Layers Using the Alpha Texture

At this stage, the crack pattern is a series of points in 2D. These co-ordinates need to be transformed in to three dimensions. As discussed in section 2.2.2, this project will generate its 3D crack pattern by linking 2D patterns together in a similar manner to [3]. The difference from [3] is the way that these extra 2D layers are created. The use an alpha texture to create the crack pattern has provided this project with another advantage. Using the technique called mipmapping. In [16], Miller describes a mipmap as "…a chain of textures with the first member of the chain being the most highly

detailed texture. For each subsequent member of that chain, the size is reduced by one power of 2." This means that by taking the first alpha texture as our most detailed, the DirectX API can lower the resolution of the texture and use that, more blocky texture, to create a lower resolution crack pattern.

It is also being considered that the number of nodes in a crack pattern should also be reduced to aid this; however, this could cause problems when it comes to linking the 2D layers together (see section 3.3.1). Lowering the resolution of the texture and crack pattern will have the effect of reducing the variation in crack pattern path which follows the physical principle outlined in [7].

3.4: Creating a 3D Crack Pattern

At the current stage of this project, the crack pattern is a two-dimensional object based upon an alpha texture. For this crack pattern to be applied to an object in 3D, the pattern itself needs to be translated in to 3D

3.4.1: Linking the 2D Patterns in to 3D

As discussed in section 3.2.1, the layers of 2D patterns need to be translated in to a single three-dimensional pattern. At present these 2D layers are simple co-ordinates in space. Due to the fact that static nodes have been used in this project, linking two layers together should not be a difficult problem.

The crack pattern created from the mipmapped alpha texture will feature longer straight lines and fewer turning points when compared to its higher resolution pattern. It is at these turning points that the link lines need to be drawn. The linking line needs to connect to its corresponding node or to the nearest node that has a crack going through it. This process is an expensive one, proportional to the number of turns in each crack pattern. Therefore, a heuristic search could be employed to detect if that turn is in fact a major turning point or a minor one (if the general direction changes for a long period of time or if the direction quickly changes again) and only the major turning points are linked. This would provide more generalised shards of the object in question and take less processing power.

An alternative to the heuristic search method would simply be to link at the origin of any crack branch and again where the branch stopped. These points could be indicated by the list of nodes themselves and would therefore take far less processing time for similar, if not better, results.

3.4.2: Crack Pattern Conversion

Once the 3D crack pattern has been created it needs to be converted into polygonal volumes. These volumes will then be intersected with the object to create the resulting meshes.

The process of making polygonal volumes from 3D crack patterns should be a simple process. By iterating through the linked nodes between the levels, polygon pieces with 6 or 8 vertices and a set of faces can be formed. These pieces form an approximation of the crack pattern, however, and do not feature the exact detail that the crack pattern has. This problem may be solved by using a more detailed linking algorithm yet this would be a trade off against processing time. However, this process is being performed offline and whilst high performance is desirable, it is not mandatory for this offline application.

3.5: Cracking the 3D Object

This section of the project is arguably the most important. Performing the cracking process and re-meshing the object confirms the visual representation of the crack pattern and validates the simulation. First, an intersection test must be found and then a clipping method must be chosen.

3.5.1: Intersection Tests

A simple test exists to check which side of a plane a point in space occurs. Figure 3.8 shows two vertices and a triangle. Vertex 1 is on the same side of the triangle as the direction of the triangle normal whilst vertex 2 is on the opposite side to the triangle normal. The position test is a simple dot product calculation. The vector between the vertex and a vertex on the triangle (the average vertex is also acceptable) is formed (the blue line) and the dot product of this and the triangle normal is calculated. If this value is less than 90° then the vertex is on the same side as the triangle normal. This process is repeated with vertex 2 and the triangle normal, though the resultant of the dot product will be greater than 90°, indicating that the vertex is not on the same side as the triangle normal.

3.5.2: Breaking the Object

The process of breaking the 3D object involves translating the crack volume to match the direction of the impact, transforming the pattern to match the force applied during the impact, calculating the resultant meshes and rendering them.

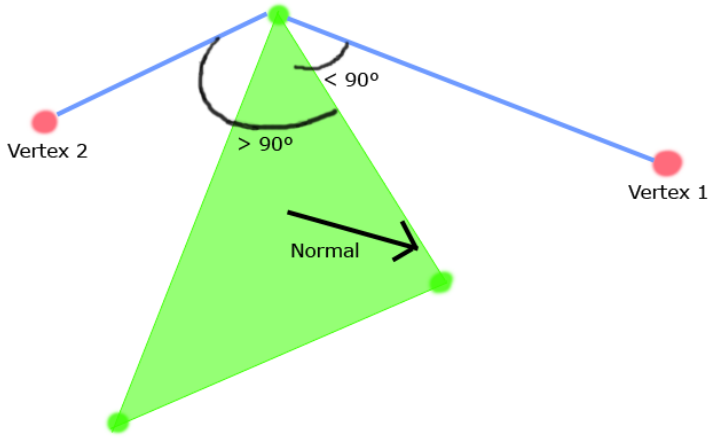


Figure 3.7 - Intersection Test for Vertices and a triangle

Calculating the resultant mesh could be done in one of three ways.

Matching Nearest Edge – In this technique the crack polygon matches its edges to the closest edges in the model. This method means that no clipping work needs to be performed but it reduces the accuracy of the crack pattern. This method was implemented by Fox in [2] with good results.

Clipping (no tessellation) – This method intersects using basic object collision and clips the mesh hard against the crack pattern, creating smooth edges that are accurate to the crack pattern. After the mesh has been clipped the vertices inside the polygon are not changed and this can lead additional polygons being drawn unnecessarily.

Clipping (with tessellation) – This is the same as above; however, after the clipping stage the object is re-meshed. This adds an extra process to the splitting algorithm yet it will render quicker afterwards.

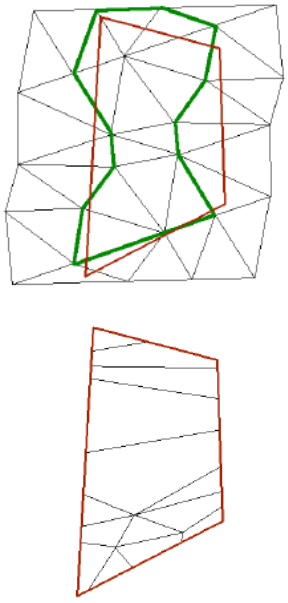


Figure 3.8 - Clipping Options

Of these three ways, the clipping with tessellation will produce the best results, however, in [2], Fox was unable to make his clipping algorithm work correctly and therefore he used the edge matching algorithm. This did produce acceptable results but the clipping algorithm is the most desirable.

3.5.3: Sealing the Meshes

By their very nature, polygon meshes are hollow with polygons simply representing their surfaces. During the cracking process this surface representation will be split into smaller pieces and will reveal the mesh's hollow nature. The process of sealing attempts to add polygons over the exposed areas so that the mesh has no holes. Sealing can be performed wherever clipping has taken place. By using the newly generated vertices from the clipping process the outline of the hole in the mesh can be found. A triangulation algorithm is then performed upon these vertices in order to seal them. [22] Presents Seidel's triangulation algorithm and [23] presents the Delaunay triangulation algorithm. Both of these techniques will seal the mesh correctly and with an optimal number of triangles. A third way is the naïve triangle fan method or an "ear-cutting" method [25] which is similar to the triangle fan but more efficient. This method is also easier to implement and requires less computational power.

Of these methods, the best algorithm would be the Delaunay triangulation but in a process where computational speed is of the essence, the ear-cutting method would

provide the most benefit for the cracking process. However, with the ear-cutting method the post-cracking animation may be slower.

3.5.4: Animating the Pieces

Finally, once the object fragments have been created and the old mesh disposed of; these new meshes must be animated away from each other. These pieces will be animated using the rigid body method described in section 2.5. Enabling collision with the surrounding world has been considered an optional addition to this project as it is not part of the main cracking process.

Summary
The choice of fragmentation algorithm is an important one and speed is of the essence here as this is the only process being performed in real-time. Therefore, the algorithm should produce the best visual results whilst maintaining highest performance. Therefore, the clipping with tessellation algorithm would be the best choice.

3.6: Program User Interface

The user interface (UI) for the crack pattern generator and for the crack viewer should ideally be clear and concise whilst allowing the full control of the variables in the environment. Globally (i.e. in both applications), the controls for the world camera should allow full rotation with click-drag mouse movements and zoom functions with the scroll wheel. This is the most common setup for object model viewers and is the default for a DirectX project. There should also be a frame rate counter displayed in the top-left corner of the screen. The following additional program specific elements are located down the right-hand side to allow adequate room to view the crack pattern.

Fracture Generator – In the fracture generator there will be a control for controlling the impact force (limited by a slider), the ability to increase the pattern resolution by increasing the grid size and to increase the number of layers. Also, it will feature the ability to load alpha textures dynamically in to the program for instant use.

Fracture Viewer – The objects will be loaded up from a file with its crack pattern library at the program start. These object names will then be loaded into a selection box for the user to pick the required object. The impact point selector will select the appropriate vertex on the object via a click of the mouse. A force will be able to be selected via a scale factor based upon the impact point and finally there will be a single button to initiate the cracking process. The cracking/animation control buttons (pause and continue) will be below the cracking initiation button on the right hand side of the screen.

3.7: Project Specification

This project should meet the following requirements for it to be considered complete:

No.	Specification	Priority
Cracking		
1	**Crack pattern should have a true three-dimensional representation**	Mandatory
2	**Fracture generator should be abstract from the object model**	Mandatory
3	**Crack pattern should depend upon an alpha texture**	Mandatory
4	**Crack pattern propagation should depend upon a specified material type**	Mandatory
5	**Simulate application of an area force instead of a point source**	Desirable
6	**Simulate multiple synchronous and asynchronous impacts**	Optional
7	**Simulate minor deformation with a crack pattern**	Optional
User Interaction		
8	**Choose the location(s) of impact point(s) on the object surface**	Mandatory
9	**User can manipulate their view of the world**	Mandatory
10	**A frame rate counter is displayed**	Mandatory
11	**User can specify the force applied during impact**	Mandatory
12	**User can adjust the speed of the cracking process and its animation**	Desirable
13	**User can specify an area for impact to occur**	Desirable
Animation		
14	**Overall frame rate should not drop below 60fps**	Mandatory
15	**Overall frame rate should not drop below 100fps**	Desirable
16	**Animate the resulting objects with a physically based path**	Desirable
17	**Animate collision between objects changing their direction**	Optional
Object Modelling		
18	**All object should be represented by continuous polygon meshes**	Mandatory
19	**Imported models should be saved in the .x (DirectX) file format**	Mandatory
20	**Hollow object should be able to be imported and cracked**	Optional
Program Code		
21	**Program code should be highly efficient and bug-free**	Mandatory
22	**Program code should be well documented**	Mandatory

Table 3.1 - Project Specification

3.8: Testing and Evaluation

Testing and evaluating the program is an essential part of the project. The visual results of the crack pattern must be able to match real-world results and be acceptable to user testing.

The main development platform for this project has an AMD Athlon 64 X2 3800+ dual core processor, 1GB of DDR RAM and an nVidia GeForce 6800GT with 256MB of graphics memory. This is an extremely powerful modern computer which should be

easily capable of performing this task in real-time. This does mean, however, that the system should be tested on lower specification platforms to make sure that it runs adequately on those computers too.

3.8.1: Testing and Evaluation

Creating a crack pattern is a large part of this project. Therefore, the crack pattern should be continually tested and evaluated. For this purpose, a test set of alpha textures has been created. During development, the crack pattern shall be tested to see if it matches the expected output for each of the alpha textures. These textures and a short description of what the result should look like can be found in appendix A.

3.8.2: User Testing

This project is not about physical accuracy of a crack pattern, it is about realistically modeling cracks using pseudo-algorithms. Therefore, the best test that can be performed to evaluate this project is with user testing.

This user test comes in the form of a plausibility test. A person is shown a set of pre-recorded fractures performed on a range of objects and asked to rate the plausibility of the fracture on a scale of 1 to 5 (where 1 is not plausible and 5 is realistic). This test would be performed on a focus group containing at least 20 people to be able to make conclusions about the results.

3.8.3: Real-World Testing

Comparison of this simulation against objects in the real world is a very important test for the project. Although the results of the simulation are not expected to match the real-world exactly, the results need to be visually similar.

Real-world testing will compare slow-motion video footage of object fracture with the computer simulation across a number of objects. This comparison footage will be given to the users as part of the user testing and they will be asked to rate the accuracy of the simulation on a scale of 1 to 5 (where 1 is poor and 5 is realistic).

Chapter 4: Design

This section documents the design of the system that will implement the theory described in chapter 3. Section 4.1 describes the tools used in the implementation, section 4.2 details the division of the system and why it was necessary do perform this division. Section 4.3 discusses the Fracture generator and section 4.4 describes the fracture viewer. Finally, the user interface is discussed in section 4.5

4.1: Implementation Language and Tools

4.1.1: Programming Language

The programming language used to implement the system will be C# [18]. Alternatives considered were C++ and Java [19], however, there were more reasons to program in C# than in the other languages.

C++ allows for very low-level access to a computer's resources by the use of pointers and is the industry de-facto standard for very high performance applications. However, the author's knowledge of this programming language is very limited and it was considered that learning this language as the system was being written would place constraints on what could be constructed in the allotted time.

Sun's Java is a high-level object-oriented programming language that runs on a virtual machine (VM) which means that the system could be executed on any operating system that supports the Java VM. However, this VM means that execution of the program is slower than a language compiled into machine specific code. Since performance is imperative in this system, Java would not be a viable option.

Microsoft's C# is a new language created for the Microsoft .NET framework. C# compiles to a "common language runtime" (CLR) which acts like a VM for the .NET framework. However, at runtime this is compiled into machine code and the program executes at speeds comparable to C++. C# has advantages over C++ in that what would take 200 lines to write in C++ only requires 50 lines of code in C#. This allows for rapid application development which is important for this project. For this reason, C# will be the language of choice.

4.1.2: Graphical Application Programming Interface

The C# programming language does not have native support for rendering 3D graphics. 2D graphics are supported through the .NET framework's GDI+ (Graphics Device Interface plus) rendering method but this is not viable for high performance graphics. There are two major Graphics Application Programming Interfaces (APIs) currently in use; OpenGL [20] and Direct3D (the 3D rendering component of Microsoft DirectX) [21]. In 2002, Microsoft released Managed DirectX (MDX) which enabled developers to write

high performance graphical applications in a managed environment. Simply put, MDX is a wrapper for regular DirectX.

All of the APIs have benefits and drawbacks but it was decided that MDX would be used in this project. This is because MDX contains many built-in types, thereby reducing the amount of basic programming that has to be completed. MDX also natively supports its own 3D model format. The .X file format contains position, normal, texture and index data about each vertex of a polygon model. MDX contains simple methods to load these files and so save a lot of time gaining compatibility with mesh file formats.

4.1.3: Other Tools

Microsoft's Visual Studio 2005 [26] was used to develop the implementation, Autodesk 3D Studio MAX 7 [27] and Milkshape 3D [28] will be used to create any 3D models that are required.

4.2: A Tale of Two Systems

From the outset of this project, creating crack patterns in real-time was always in question; this was especially relevant due to the manner in which these crack patterns were to be created. The path-finding algorithm which directly referred to a texture to find the path of least strength for a crack could not be performed without a noticeable delay in the rendering of the next frame. This was especially noticeable once additional algorithms were implemented to convert this crack pattern to a 3D mesh. Such an operation could take up to 5 seconds for a low resolution pattern even on the most powerful home computers. An interruption to scene rendering of this magnitude severely interrupts the flow of the game play and is therefore an unacceptable solution.

There were two solutions considered to this problem. The first was to reduce the complexity of the crack pattern. This solution would create a simple crack pattern based upon mathematical shapes rather than a texture. Crack patterns generated in this style would be simple in shape, representing a rather generic "spider's web" style of pattern, but would be easy enough to generate in real time. However, this method would take no account of a 3D object's strength texture and could therefore create inaccurate results over very strong areas of an object.

The second solution to this problem was to separate the crack pattern generation into a separate process that could be run offline. The crack patterns generated from this would then be stored in a library that is loaded in to a second program which performs the cracking itself. At runtime, this program would pick the most appropriate crack pattern from the library and then apply it to the target. This allows strength textures to be used and also allows for more complex and more accurate crack patterns to be created as the crack generation process is now time independent.

It was decided that the second solution would be more effective for this project as this will allow for the highest performance with the most detail present in the crack patterns.

4.3: Fracture Generation

This section reviews the design of the Fracture generator system. This program generates crack patterns based upon a strength texture, creates a 3D pattern from this and then saves this 3D pattern as a .X format mesh.

4.3.1: Fracture Generator Components

The fracture has many individual classes that link together to form the whole application. Each class has a particular function and these functions are described below.

Renderer
This class handles the rendering of the scene. It used the MDX API to render the scene that is presented to it. It also contains the user interface controls.

CrackPattern3D
This class holds an array of CrackPattern objects. Each of these crack patterns represents a layer of the crack pattern itself. This class contains methods to create these layers from the first layer and also to create a mesh from these layers. These meshing techniques are discussed in section 4.3.3

CrackPattern
This class holds the branches and roots of the crack pattern itself. It contains methods to generate these roots and branches from an alpha texture which is passed to this method. This class contains the method which retrieves the pixel data from the alpha texture. A more detailed look at this class is found in section 4.3.2

Crack, CrackRoot and CrackBranch
The above three classes are intrinsically linked with each other and are the core of the crack pattern generation system. They are explained in detail in section 4.3.2

4.3.2: 2D Crack Pattern Generation

The process of 2D crack pattern generation is controlled by the CrackPattern class. The crack pattern is generated in 2D before being transformed into a 3D model because the alpha texture that it is given is in a 2D form itself. Having a 3D crack pattern would be beneficial for the realism of the fracture; however, a crack pattern propagating in three dimensions would be much harder to control and gain reproducible results. 3D textures also contain a much larger amount of data than their 2D counterparts and, in general, are made procedurally so that they do not consume as much memory.

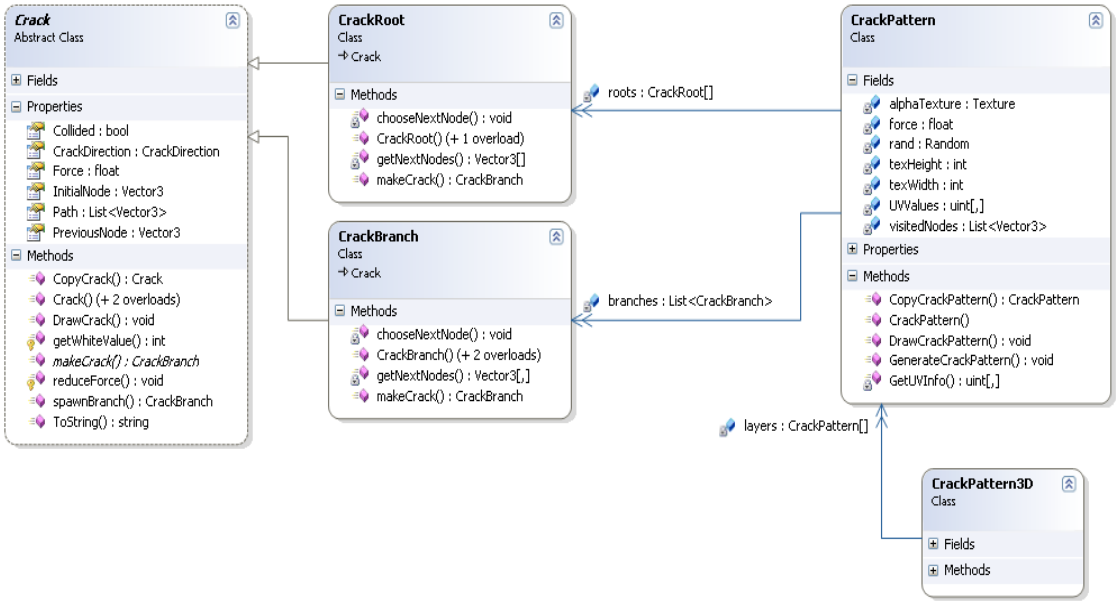


Figure 4.1 - Fracture Generator class diagram

The command to create a crack pattern is handled by the CrackPatten class, detailed below.

CrackPattern

As described in section 4.3.1, this class holds the data for a crack pattern. This data is an array of 4 CrackRoot objects, a list of CrackBranch objects and the pixel colour data for the alpha texture supplied to it. The principal task of this class is to create a crack pattern using the alpha texture, achieved through its generation method.

Crack

Crack is an abstract class. This means that it contains fields and methods that its sub-classes use but also contains signatures that must be implemented by the sub-classes. Crack contains fields for the path of the crack pattern, impact location information and a general direction that the crack is to travel in. This direction is very important as it is used to determine whether the next point chosen by the crack is valid. This is to prevent a crack doubling-back on itself and aids the propagation of the crack pattern.

Crack's methods relate to general functions for the roots and branches of the crack pattern, these include the process of spawning a branch, reducing the amount of force in the crack and finding the strength of a node. Crack contains only one abstract method, makeCrack, which calculates the next point to go to. This is implemented in different ways by CrackRoot and CrackBranch. Details of the implementation can be found in section 5.2

4.3.3: Converting a 2D Crack Pattern to a 3D Mesh

In order for a crack pattern to be used in the real-time element of this system, it must be converted into a polygon mesh. Therefore, it is necessary to convert the 2D crack pattern in to a 3D mesh.

The conversion of a pattern from 2D to 3D was described in section 3.3, however, further constraints were added to the way that the mesh must to be constructed.

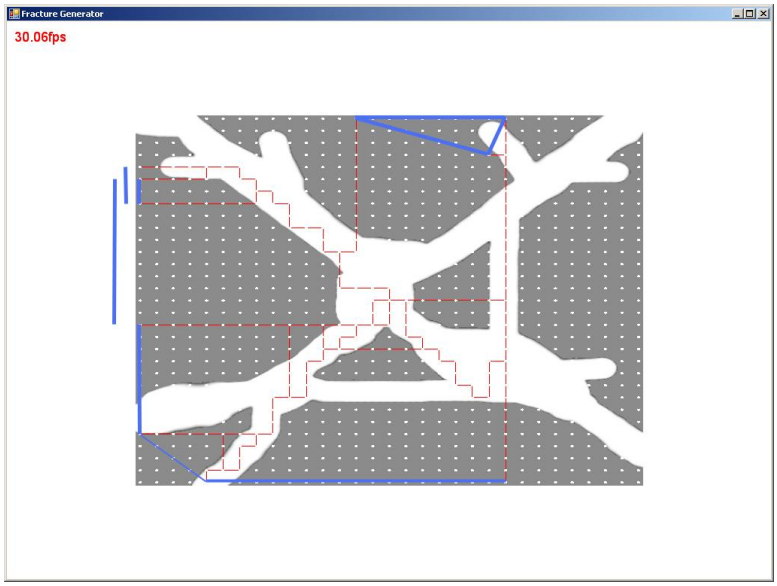


Figure 4.2 - Endpoint Meshing, the blue lines represent the edges

- ¶ In order that each volume does not have any open sides, the crack pattern must be amended so that each volume is bound by an edge. This edge can be diagonal.
- ¶ So that each individual volume in the mesh can be identified, during triangulation each volume must have its triangles indexed consecutively in the index buffer. The production of a mesh will also produce a file containing the first and last index of each volume in the index buffer.

These two problems can be classified as enclosing and volumization respectively. The final stage of constructing a 3D crack pattern mesh involves the triangulation of the 2D volumes.

4.3.3.1: Enclosing

In order to comply with the first restriction, set out in section 4.3.3, a number of methods were considered:

Endpoint Meshing

This technique involved locating the end points of each of the roots and branches and joining them together by finding the closest endpoint to that endpoint. This method produced unsatisfactory results as end points were often an equal distance apart or that the correct join would be a large distance away and another end point would take priority. A result of this method, with all of the joins drawn in, can be found in figure 4.2

Shrink-wrap Meshing

Using the discrete nature of the crack pattern, the method sweeps grid from each side and finds the position of the path closest to that edge of the grid. Joining these positions together creates an outline that completely seals the crack pattern's volumes. Figure 4.3 provides a good example of a shrink-wrapped crack pattern.

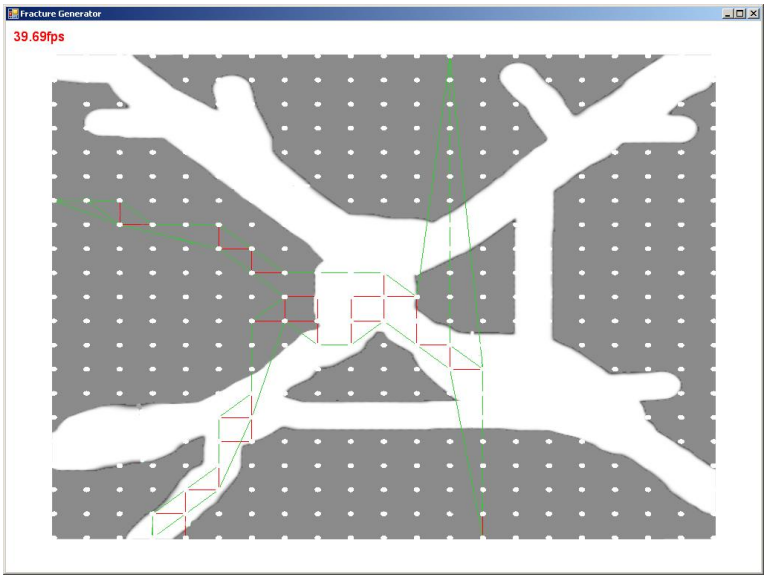


Figure 4.3 - Shrink-wrap meshing

However, as can been seen from figure 4.3, whilst the method produces an enclosed pattern (green lines), it does not produce convincing volumes which can be used for intersection in the real-time application. The volumes that it produced hug the crack pattern and are not suitable for application.

Clockwise Meshing

This technique is similar to that detailed in the endpoint meshing method in this section. The method retrieves the endpoints of the crack pattern's branches and uses simple trigonometry to calculate the angle between the impact point of the crack pattern and each endpoint. These values are then sorted in ascending order. It follows that a pattern can be successfully sealed by joining each of the endpoints in the order of sorted angles. This joins the endpoints in a clockwise fashion around a circle, hence the name of the technique.

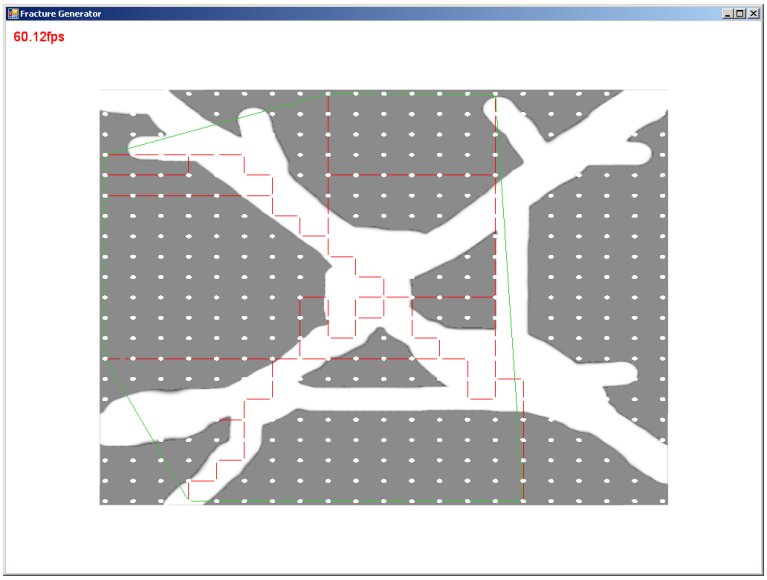


Figure 4.4 - Clockwise meshing

The results of this method (seen in figure 4.4) were very good with maximal volumes being formed from the endpoints. However, a close look at the right hand side of figure 4.4 shows one of the bordering edges crossing the line of one of the branches. This small error would require the manual editing of the crack pattern before use in the real-time simulation. A

more balanced crack pattern, i.e. one that has branches extending completely to each corner, would not exhibit this behaviour.

The clockwise meshing technique is by far the most successful technique and was therefore implemented into the design of the system.

4.3.3.2: Volumization

Volumization is the process of locating the boundaries of each enclosed area. To a human this process is trivial; however, to a machine it is much more difficult as the machine knows nothing of a visual shape, only a number of joined points. Two methods were considered for this task:

Volume Sweeping
This technique uses the discrete nature of the crack pattern to effectively sweep the crack pattern for edges, constructing volumes from these edge locations.
The sweep begins at any side, ignoring any boundaries created by the enclosing procedure. Separating each volume into the number of discrete points between each horizontal line at the start of the sweep, it then proceeds to sweep across the crack pattern until it reaches a section of vertical branch in the crack pattern. This collision is recorded and a new area of volume is constructed so that the sweep may continue. Once the sweep ends, these areas are reconstituted into their complete areas. This method is very costly, an order of magnitude > N^2 and therefore, other methods should be considered.

Volume Tracing
Volume tracing is based upon the assumption that each volume is completely sealed and therefore, from any starting point where a branch starts or collides with another branch, following this path in a certain direction will always lead you back to the first point. Starting at the impact point, this technique traverses along its path until it finds a collision or a start point. After considering whether this new path is travelling in the desired direction (set as 'left' in the implementation) this technique then traverses the path that is going in the correct general direction. This procedure is then repeated until the position that the volume tracer started at is reached again. This path is the outline of one of the volumes. This process is then repeated for each branch. Once completed, any duplicate volumes are discarded and the paths are ready to be converted in to 3D and triangulated.
This method is also difficult to implement but it is less expensive than the volume sweeping technique detailed above.

4.3.3.3: 3D Transfiguration and Triangulation

In section 3.3, it was suggested that a 3D crack pattern was to be a set of 2D crack patterns with each layer being a less detailed version of the previous layer. This feature would add an extra level of detail to the crack pattern but it was initially considered that

before this was implemented that a simpler crack pattern should be created to gauge whether the implementation worked. Therefore, a 3D crack pattern was created as a set of 3D crack patterns with each layer being the same as the first one.
These layers are joined at every point along the crack pattern's branches, each join running from its XY coordinate to the same XY coordinate on the next layer. This provides for some very simple triangulation. Unfortunately, due to implementation problems with the volumization algorithm, the additional layers of detail were not implemented.

Triangulation in this project would take place upon each of the layers and inside the links between the planes. This triangulation would take place upon each volume of the mesh in turn so as to provide the optimal mesh for intersection in the real-time application. This means that some triangles will be indexed twice, though their winding rules will differ, thus producing a double-sided triangle. This is necessary for the clipping procedure to function correctly (see section 4.4.2).

Three techniques were considered for triangulation, these were Seidel triangulation [22], Delaunay triangulation [23] and a naïve triangulation method. The naïve method uses the discrete grid to form a large number of small squares which are trivially triangulated. Due to time constraints, the system of naïve triangulation was implemented into the design of the generator.

4.4: Fracture Viewer

The term "fracture viewer" refers to the real-time system that performs the division of the target mesh. This program loads textures and crack patterns into its memory to form a library. A crack pattern is then chosen from this library to fracture the target polygon mesh.

4.4.1 – Fracture Viewer Components

Renderer
This class acts in the same way as the renderer described in section 4.3.1

FractureViewer
This class handles all the commands passed to it by the renderer. FractureViewer also loads the CrackLibrary and controls the rendering state of the world and the target mesh inside the world.

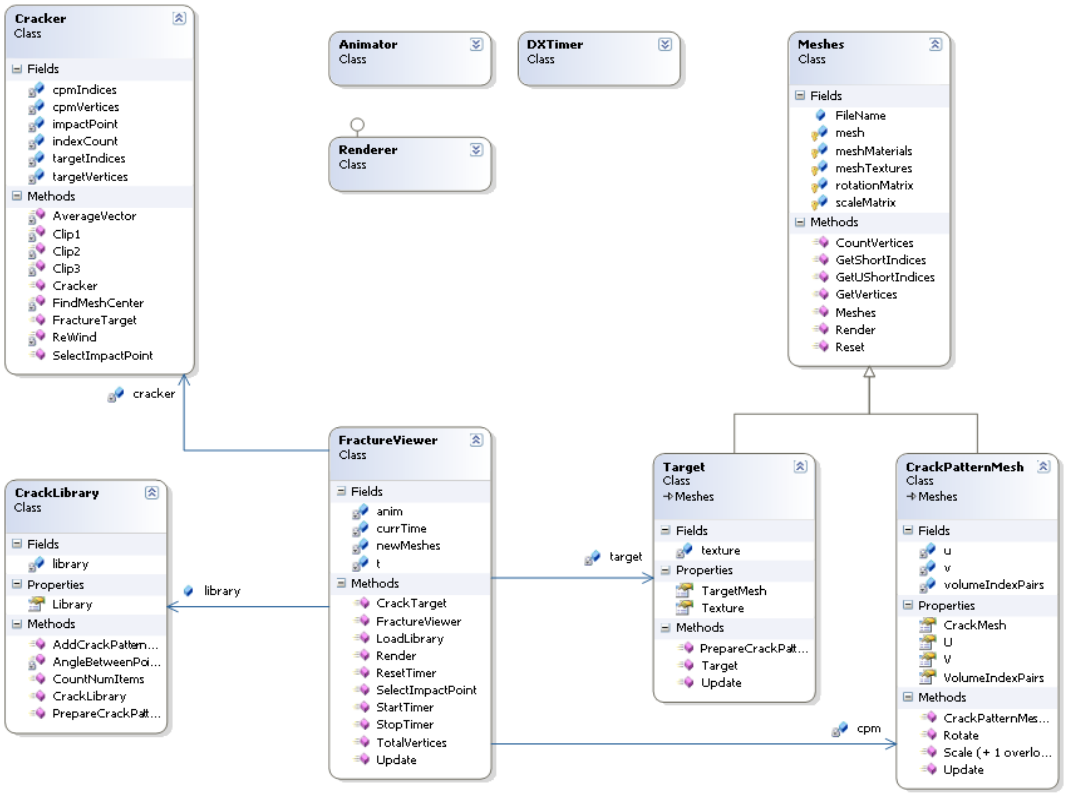


Figure 4.5 - Class diagram for Fracture Viewer

CrackLibrary

A CrackLibrary holds a number of (Texture, List<CrackPatternMesh>) pairs. The texture refers to an alpha texture and the list of CrackPatternMesh objects is a set of many different crack patterns created using that texture. The CrackLibrary holds all of the textures in the test set (Appendix A) and 5 crack patterns generated from each of these (each crack pattern being generated from a different impact point on the texture, one in the middle of each quadrant and one at the centre of the texture).

Once the user chooses an impact location on the target, the CrackLibrary picks the crack pattern that was generated from an impact point closest to that of the user's choosing and that is associated with the target's texture. The CrackLibrary is then responsible for orienting the crack pattern in accordance with the direction of the impact.

Meshes

Meshes is a simple class that performs basic functions upon a mesh object. These functions include loading from a .X file, rendering and retrieving the vertex and index buffers. Meshes acts as a base (abstract) class for the Target and CrackPatternMesh Classes.

CrackPatternMesh

A CrackPatternMesh inherits the methods and properties of the Meshes class. It contains a number of properties, these include a coordinate for its impact location and a Dictionary<short, short> of volume indices. These volume indices indicate the first and

last index of each volume in the particular crack pattern. These are obtained from a text file upon loading the crack pattern into memory.

Target
A Target is similar to a CrackPatternMesh in that it inherits from Meshes; however, its only additional property is a texture that it is associated with. During the placement of an impact point by the user, the Target sends its texture data to the CrackLibrary so that it may choose from the correct series of CrackPatternMesh objects.

Cracker
The Cracker class controls the placement of impact point and the intersection of CrackPatternMesh and Target. This class is discussed in detail in section 4.4.2

Animator
This class controls the animation of each mesh after the cracking process has taken place. Animator is designed to have two methods, one for linear animation and another for kinematic animation. These two methods are interchangeable for ease of use.

DXTimer
This class retrieves a high-precision system time and also allows for advanced control of this timer. This modified from an example given in [16].

4.4.2: Cracking and Clipping

The primary function of the fracture viewer program is intersection and reforming of the target polygon mesh. This functionality is provided entirely by the Cracker class. There are two public methods in this class; SelectImpactPoint and FractureTarget. These two functions are discussed below.

Select Impact Point
In order that the user may choose an impact location upon the target mesh, this method uses a ray tracing method to determine the world-space coordinates of a mouse-click. A ray is fired from the camera in to the scene and an intersection test is performed between the ray and the Target object. If intersection is successful, the world-space coordinates are given to the CrackLibrary which loads and orients the CrackPatternMesh as described in section 4.4.1

Fracture Target
The test for intersection between a polygon and the volume was discussed in section 3.4.1 whilst the choice of clipping procedure is described below.

Clipping Procedure
Of the three techniques detailed in section 3.4.2, it was decided that clipping with tessellation should be implemented. This is because it will produce the best looking

result with the best post-cracking efficiency. The nearest edge method would provide better system performance during the cracking though has the potential to produce unrealistic results if the target mesh has large polygons. The aim of this project is to provide for a realistic approximation of material fracture and the nearest edge method would not provide for this.

During the clipping process, new polygons must be created at the clipped plane. These polygons will be triangular for the following reasons:

- ¶ Modern 3D graphics accelerators render triangles faster than any other type of polygon
- ¶ Any unclipped polygons will be triangular. Mixing polygons is not advisable
- ¶ The Direct3D API does not support polygons with more than three edges unless it is rendering a patch based mesh.

4.5: User Interface Design

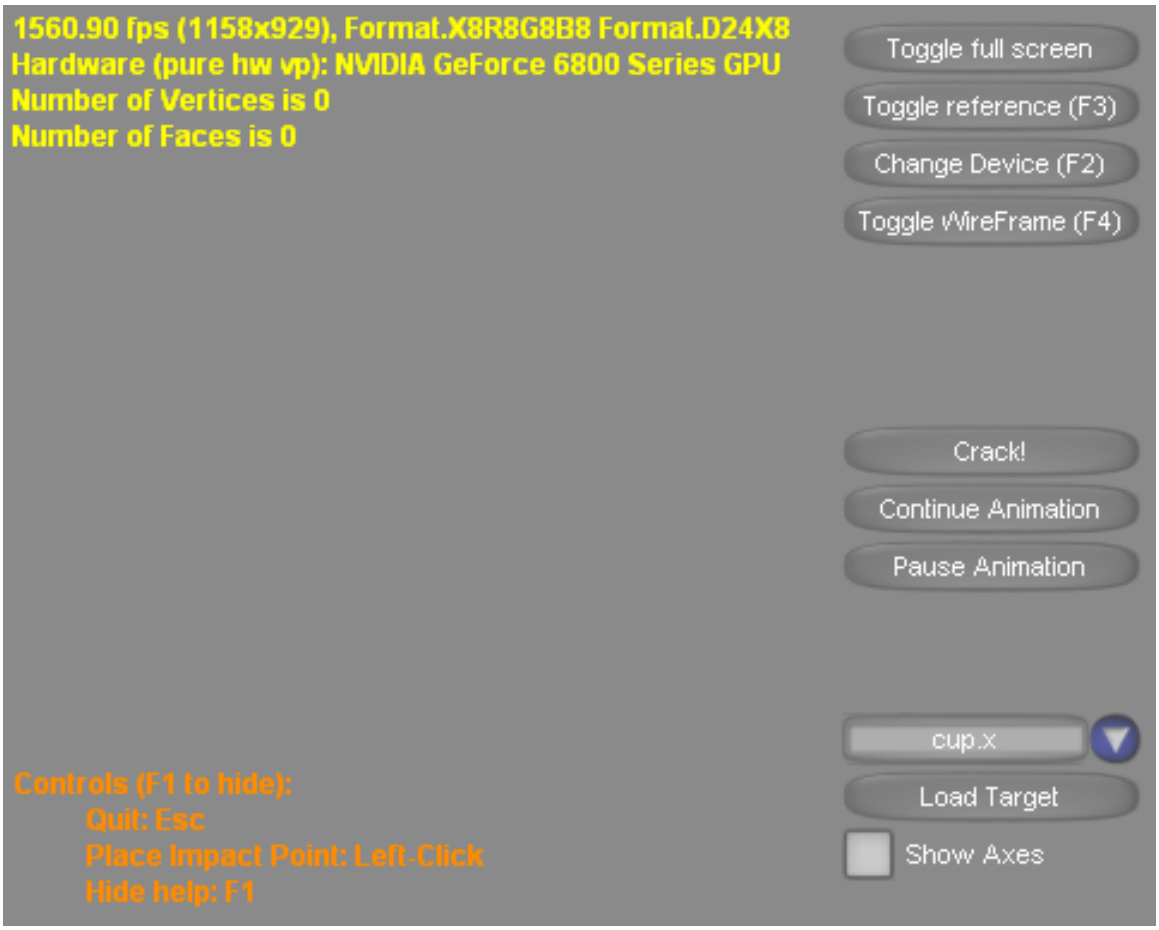


Figure 4.6 - Fracture Viewer User Interface

The user interface for the fracture viewer was based upon the DirectX EmptyProject sample. This sample includes a high precision frame rate counter, detailed information about the graphics device present in the computer and the ability to change to a

reference rendering device (if a graphics card does not possess a required feature). The user interface for the Fracture viewer can be seen in figure 4.6.

In addition to the aforementioned features, the user interface (UI) features buttons on the right-hand side to draw the scene in wireframe mode and to initiate the cracking of the mesh. Grouped with the cracking button are controls for the post-cracking animation. Pressing the button labelled ‘pause animation’ ceases the movement of the resultant pieces of the mesh. A click on the button labelled ‘continue animation’ restarts the animation from where it was stopped.

The bottom left-hand corner displays helpful information about the application including how to place an impact point. This information can be hidden by pressing the ‘F1’ key. The application can be closed by pressing the ‘Esc’ key or clicking on the x in the application’s menu bar.

The fracture generator features a much simpler UI. The fracture generator responds to a set of key presses on the keyboard to perform all of the actions. The arrow keys control camera position, WASD keys control the camera yaw and roll and the space and L-Ctrl control keys control the camera pitch. The cracking process is started by pressing the ‘C’ key. 3D layers are formed by pressing the ‘L’ key, these layers are joined together by pressing the ‘J’ key and the pattern is enclosed by pressing the ‘O’ key. The mesh is volumized and triangulated by pressing the ‘V’ key. The fracture generator is set up in this manner as during the survey and analysis section of the project this simple interface was constructed to prototype the crack pattern generation. After this process development continued using the same interface and no reason was seen to change it until development had ceased. Due to problems with the algorithm for volumization, the fracture generator was not fully completed and therefore was not moved to a new interface.

Chapter 5: Implementation

This chapter describes the implementation of the system described in chapter 4. Programming snippets in this section detail the innovative components of this system. The final section in this chapter details any problems encountered and compares the implementation to the specification set out in section 3.6

5.1: Programming Language Semantics

The code examples in this chapter were written in the programming language C# 2.0. This language has some specific tags not found in other languages.

- The < > declaration after a collection type indicates what type of data is to be held in this collection. This is a technique called generics and allows for type-checking at compile-time without the need for a cast to that type whenever the collection is accessed.
- The ? operator is a ternary operator. It functions like a single line *if then else* statement i.e. *expr1* ? *expr2* : *expr3* reads as *If* expr1 *then* expr2 *else* expr3

Generic declarations are very important in C# performance. Due to the .NET CLR, all value types in C# are subject to 'boxing' i.e. converting to an Object type before being added to memory. To access the variable again it has to be 'unboxed'. This process requires additional resources. However, declaring the type of a collection does not then require each of the types in a collection to be 'boxed' thus saving a lot of processing time which is essential for the high-performance of this project.

5.2: Crack Pattern Generation

5.2.1: Parameters

A crack pattern requires a number of parameters in order for crack pattern generation to occur. Each of these parameters has a significant effect on the result of the crack pattern. These parameters and their effects are detailed below:

- *Force* – This property relates to the amount of force the target is struck with. The implementation of this property effects the distance a crack will travel before stopping. This is especially important in situations where there are a large number of *rows* and *columns* in the crack pattern as a higher force will be required to allow the crack pattern to reach the edges of the grid.

- *Rows* and *Columns* – These properties control the size of the discrete grid that the cracks navigate around. Larger values of rows and columns produce a higher-resolution crack pattern when combined with a higher force value.

- *Number of Layers* – This controls the number of layers in the crack pattern. Increasing this value will increase the depth of the crack pattern and the number of volumes created.

- *Alpha Texture* – This value changes the alpha texture that is being used to generate the crack pattern. Any texture can be used but a test set (see Appendix A) has been provided in the application. The alpha texture controls the direction of the crack pattern and its branches and has the highest influence upon the final look of the crack pattern.

5.2.2: Crack Pattern Algorithm

The generation method creates 4 CrackRoot objects and orders them in turn to find their next destination which is determined by the CrackRoot algorithm (section 5.2.3). The method continues to send this order, reducing the overall amount of force after each iteration of the loop. The generation method checks to see if any CrackBranch objects have been spawned during this process and adds them to its data storage. These branches are then iterated in the same way as the roots of the crack pattern, however, the destination determination algorithm for a CrackBranch is different to that of CrackRoot and is detailed in section 5.2.4.

Destination determination is performed in this “round-robin” fashion as allowing cracks to grow until their natural termination in turn will not produce accurate results because many cracks will terminate abnormally i.e. colliding with the path of another crack.

Once all of the branches and roots have had their forces reduced to zero, CrackPattern stores the branches and roots in its data and then passes this data to the renderer.

5.2.3: CrackRoot Algorithm

This class is a sub-class of Crack and so inherits all of its properties and methods (see section 4.3). It contains an implementation of makeCrack which determines the next destination of the crack. In CrackRoot this is a simple implementation performing the following operations:

- Taking its current position, CrackRoot looks in the three directions that it travel in and finds their positions, storing these positions in an array.
- CrackRoot then calculates the strength value at each of these positions and passes the position with the lowest strength value (highest white value) to check if this position is valid
- A position is valid if the direction that the new position is in is available to that crack.
- The new position is then tested to see if it has reached the edge of the pattern or if it has collided with another crack. In either case the remaining force in the crack is then set to zero. If this does not occur, the force is reduced

proportionally in accordance to the strength of the material that is it passing through.

¶ Finally, the new position is added to the crack's path.

Figure 5.1 shows C# code for this algorithm.

```
private void chooseNextNode(Vector3[] nextNodes, Vector3[] vn)
{
    Vector3 chosenNode = nextNodes[0];
    bool nodeAcceptable = false;
    bool rejected = false;

    while (nodeAcceptable == false)
    {
        int randomtemp = r.Next(30);
        int random = (int)randomtemp / 10;

        // Check if the node has already been visited on the path
        if (!path.Contains(nextNodes[random]))
        {
            for (int i = 0; i < nextNodes.Length; i++)
            {
                if (!rejected)
                {
                    if (getWhiteValue(nextNodes[i]) > getWhiteValue(chosenNode))
                        chosenNode = nextNodes[i];
                }
                else
                {
                    chosenNode = nextNodes[random];
                }
            }
            nodeAcceptable = true;
        }

        // Check to make sure the crack is going in the right direction
        Vector3 current = (Vector3)path[path.Count - 1];
        nodeAcceptable = ValidateDirection(chosenNode, current);
       if (!nodeAcceptable)

            rejected = true;
    }
    // set previous node!
    previousNode = path[path.Count - 1];

    // and add that node to the path
    path.Add(chosenNode);

    // check to see if the chosen node is at the edge
    // If it is then reduce the force to 0
    if (chosenNode.X == xLimit || chosenNode.X == -
xLimit || chosenNode.Y == yLimit || chosenNode.Y == -yLimit)
    {
        force = 0;
    }
    // check to see if the chosen node has been visited by another crack
    // If it is then reduce the force to 0
    foreach (Vector3 v in vn)
    {
        if ((chosenNode.X == v.X) && (chosenNode.Y == v.Y))
        {
            force = 0;
            collided = true;
        }
    }
    // reduce the force accordingly
    reduceForce(getWhiteValue(chosenNode));
}
```

Figure 5.1 - C# code for FindNextNode

5.2.4: CrackBranch Algorithm

This class is a sub-class of Crack and so inherits all of its properties and methods. It also contains an implementation of makeCrack. In CrackBranch the implementation is more complex; it shares the same validity and force reduction tests as CrackRoot but its position calculation test is different. It adheres to the following structure:

- ¶ Taking its current position, CrackBranch looks in the three directions that it can travel in and then looks a further position ahead in each direction. It stores this data in a two-dimensional array, the first element containing the position data of the closest of the two positions and the second element containing the position data of the further of the two positions.
- ¶ CrackBranch then retrieves the strength values of these positions and creates a score for each direction. CrackBranch then chooses the direction which is second best. In section 3.2 it was found that a branch would choose to find its way back to the branch it came from as this is the weakest route. Therefore the second-best route should be taken.
- ¶ CrackBranch then performs the validation procedure as in CrackRoot.

It should be noted that if a position was to fail the validation check then the next-best route was to be taken until a valid route could be found.

5.2.5: Mesh Creation

As discussed in section 4.3, once the crack pattern has been created it is required that the crack pattern be transformed into a mesh and saved to a file. This process requires three stages: enclosing, volumization and triangulation.

Enclosing

Of the methods described in section 4.3.3, it was decided that Clockwise Meshing was to be implemented. Figure 5.2 shows C# code for the implementation.

```
private void ClockwiseMeshing()
{
    List<Vector3> endPoints = this.getEndPoints();
    List<List<Vector3>> listOfEndPoints = new List<List<Vector3>>();
    int splittingIndex = endPoints.Count / (layers.Length - 1);
    for (int x = 0; x < layers.Length - 1; x++)
    {
        List<Vector3> temp = new List<Vector3>();
        for (int i = x * splittingIndex; i < splittingIndex * (x + 1); i++)
        {
            temp.Add(endPoints[i]);
        }
        listOfEndPoints.Add(temp);
    }

    List<Vector3> lines = new List<Vector3>();
    int linesindex = 0;
    foreach (List<Vector3> lis in listOfEndPoints)
    {
        SortedDictionary<double, Vector3> angles = new SortedDictionary<double, Vector3>(
);
```

```
        foreach (Vector3 v in lis)
        {
            try
            {
                angles.Add(this.AngleToPoint(new Vector3(), v), v);
            }
            catch (ArgumentException)
            {
                angles.Add(this.AngleToPoint(new Vector3(), v) + (Math.PI / 180), v);
            }
        }
        IDictionaryEnumerator de = angles.GetEnumerator();
        while (de.MoveNext())
        {

            lines.Add((Vector3)de.Value);
        }
        lines.Add(lines[linesindex*(lis.Count+1)]);
        ++linesindex;
    }
    lines = DuplicateSpliceVertices(lines);
    meshLines = lines.ToArray();
}
```

Figure 5.2 - C# code for the clockwise meshing algorithm

Volumization

Section 4.3 detailed the algorithm for identifying individual volumes in a crack pattern. The implementation of this algorithm followed this structure:

```
FOR EACH branch
        FOR EACH point on that branch's path
                Check for collision with another branch
                IF a collision was found
                        WHILE the current point is not the point that started the loop
                                Follow the path
                                Check for collision with another branch
                                IF a collision was found
                                        IF the new path travels in the correct direction
                                                Set that path to be the current path
                                        END IF
                                                Repeat the while loop
                                END IF
                        LOOP
                END IF
                Record the path and add it to the list of volumes
        LOOP
LOOP
Remove duplicate volumes
```

During the implementation stage of this project, this algorithm was implemented into the Fracture generator program. However, this implementation was unsuccessful in producing volumes from the crack pattern. This was due to errors in the crack pattern as often the visual representation did not represent all of the data that was contained within each crack pattern. This made it extremely difficult to diagnose problems within the algorithm. As a result of this, neither the Delaunay triangulation nor naïve triangulation methods were implemented.

5.3: Crack Pattern Application

The cracking process is triggered by the user clicking on the UI button marked “Crack” once an impact point has been placed. This calls the method in FractureViewer called CrackTarget(). This creates an instance of the Cracker class and calls the FractureTarget (Target t, CrackPatternMesh cpm) method which returns a Dictionary<Mesh, Vector3> objects which contain the mesh and its initial direction after cracking. Figure 5.3 shows code listing for FractureTarget

```
public Dictionary<Mesh, Vector3> FractureTarget(Target t, CrackPatternMesh cpm)
{
    targetVertices = t.GetVertices();
    targetIndices = t.GetShortIndices();
    cpmVertices = cpm.GetVertices();
    cpmIndices = cpm.GetUShortIndices();

    Dictionary<short, short> indexPairs = cpm.VolumeIndexPairs;
    ICollection<short> keys = indexPairs.Keys;
    List<List<CustomVertex.PositionOnly>> vertsList = new List<List<CustomVertex.Position
Only>>();
    List<List<short>> indicesList = new List<List<short>>();

    foreach (short start in keys)
    {
        List<short> indices = new List<short>();
        List<CustomVertex.PositionOnly> verts = new List<CustomVertex.PositionOnly>();
        indexCount = 0;
        for (ushort i = 0; i < targetIndices.Length; i += 3)
        {
           // This is an efficiency measure. If a triangle is all inside one volume it ne
ed not be considered again
            // If it's all in, set those indices to -1
            if (targetIndices[i] != -1)
            {
                bool v0in = true, v1in = true, v2in = true;
                List<ushort> v0 = new List<ushort>();
                List<ushort> v1 = new List<ushort>();
                List<ushort> v2 = new List<ushort>();

                for (int j = start * 3; j < (indexPairs[start] * 3); j += 3)
                {
                    Vector3 triangleNormal = cpmVertices[cpmIndices[j]].Normal;
                    Vector3 connected0 = cpmVertices[cpmIndices[j]].Position -
 targetVertices[targetIndices[i]].Position;
                    Vector3 connected1 = cpmVertices[cpmIndices[j]].Position -
 targetVertices[targetIndices[i + 1]].Position;
                    Vector3 connected2 = cpmVertices[cpmIndices[j]].Position -
 targetVertices[targetIndices[i + 2]].Position;

                    if (Vector3.Dot(connected0, triangleNormal) > 0.0f)
                    {
                        v0in = false;
                        v0.Add((ushort)j);
                    }
                    if (Vector3.Dot(connected1, triangleNormal) > 0.0f)
                    {
                        v1in = false;
                        v1.Add((ushort)j);
                    }
                    if (Vector3.Dot(connected2, triangleNormal) > 0.0f)
                    {
                        v2in = false;
                        v2.Add((ushort)j);
                    }
                }
                // Perform Clipping
```

```
                SelectClip(v0in, v1in, v2in, i, i + 1, i + 2, v0, v1, v2, verts, indices,
 out verts, out indices);
            }
        }
        if (clippedIndices.Count != 0)
            indices = SealMesh(indices, clippedIndices, verts);
        vertsList.Add(verts);
        indicesList.Add(indices);
        clippedIndices.Clear();
    }

    // Create meshes
    return MakeMeshes(vertsList, indicesList);
}
```

Figure 5.3 - C# code for FractureTarget

This algorithm contains 3 distinct parts. Firstly, each vertex is tested against each triangle in the crack pattern using the intersection test supplied in section 3.4. If any of these tests fail a Boolean value for that vertex will be set to false. Secondly, once the tests have finished, the results and the indices of each of the vertices are passed to the clipping method which is discussed further into this section. Finally, once each crack pattern volume has performed its intersection tests, the new meshes are created and returned to the FractureViewer object for rendering.

Clipping

As detailed in sections 3.4 and 4.4, a tessellation clipping method was to be implemented. The clipping method is invoked after every triangle is processed in the FractureTarget() method. The SelectClip() method determines the correct style of clipping for the triangle, code for which can be found below in figure 5.4

```
private void SelectClip(bool v0in, bool v1in, bool v2in, int index1, int index2, int inde
x3, List<CustomVertex.PositionOnly> verts, List<short> indices,
out List<CustomVertex.PositionOnly> v, out List<short> i)
{
    if ((v0in & v1in & v2in) == true)
    {
        NoClip(index1, index2, index3, verts, indices, out verts, out indices);
    }

    else if ((v0in | v1in | v2in) == true)
    {
            if ((v0in && !v1in && !v2in) || (!v0in && v1in && !v2in) || (!v0in && !v1in &
& v2in))
            {
                int insideIndex = v0in == true ? 0 : (v1in == true ? 1 : 2);
                switch (insideIndex)
                {
                   case 0:
                   Clip1(index1, index2, index3, verts, indices, out verts, out indices);
                   break;

                   case 1:
                   Clip1(index2, index3, index1, verts, indices, out verts, out indices);
                   break;

                   case 2:
                   Clip1(index3, index1, index2, verts, indices, out verts, out indices);
                   break;
                }
            }
            else
```

```
            {
                int outsideIndex = v0in == false ? 0 : (v1in == false ? 1 : 2);
                switch (outsideIndex)
                {
                   case 0:
                   Clip2(index1, index2, index3, verts, indices, out verts, out indices);
                   break;

                   case 1:
                   Clip2(index2, index3, index1, verts, indices, out verts, out indices);
                   break;

                   case 2:
                   Clip2(index3, index1, index2, verts, indices, out verts, out indices);
                   break;
                }

        }
    }
    v = verts;
    i = indices;
}
```

Figure 5.4 - C# code for SelectClip

Based upon the number of vertices that failed the intersection test the clipping method is determined. There are 4 clipping methods:

- *NoClip* indicates that all vertices are inside the volume and therefore no clipping is required. 1 polygon is created from this method with 3 new vertices.
- *Clip1* indicates that 1 vertex is inside the volume. This method adds 1 polygon to the mesh with 3 new vertices
- *Clip2* indicates that 2 vertices are inside the volume. This method creates 2 polygons and 4 new vertices.

A third clipping case can also occur; *Clip3* indicates that all 3 vertices failed the intersection test at some point. Multiple planes would be created based upon the crack pattern mesh triangles that caused the failure of the intersection. This is a very rare case but can occur with very complex crack patterns or target meshes with large polygons. However, this case is so rare that the benefit of an additional polygon on a very small amount of models did not outweigh the performance hit of an additional condition check for each polygon and therefore this method was not implemented. Full details for each of these clipping methods can be found in Appendix C.

5.4: Problems with Implementation

Fracture Generator

The Fracture generator could not produce mesh objects as the implementation of the volumization technique did not successfully identify the volumes in the crack pattern.

Fracture Viewer

During the clipping stage of the cracking algorithm, a problem with floating point precision was found. This occurred when a point was intersected against a plane to find the clipping point. The result of this was that a larger number of vertices were created. A solution to this problem was to weld the vertices with an epsilon value. Welding looks at each vertex in a mesh and any vertices that are within an epsilon value of the vertex are merged into one vertex. This epsilon is a floating-point value that indicates the lenience in merging vertices together.

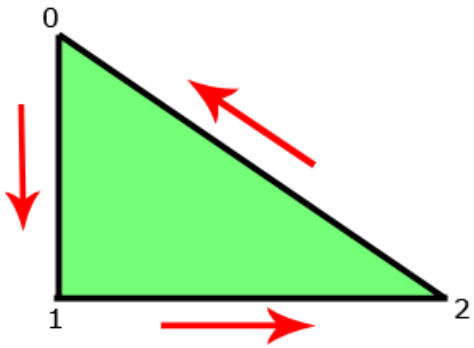


Figure 5.5 - Anti-clockwise winding

Late in the implementation of this system, a sealing method was implemented to create solid meshes. This method was a very fast clockwise triangle-fan method that traded mesh efficiency for computation time (as can be seen in the performance charts in section 6.3). However, this clockwise method came into winding order problems. As the method was performed clockwise, it follows that if a plane cracked a mesh, one resultant mesh would be correctly sealed but the other would have its seal wound incorrectly. A quick solution to this was to "double-wind" the vertices, effectively creating the seal twice for each resultant mesh. This is a fast method for solving the problem but it is very inefficient. A better way would be to perform either Delaunay triangulation or the ear-cutting algorithm upon the incomplete area of the mesh. However, this would still produce the winding problem.

5.5: Specification Fulfillment

Table 5.1 shows the specification laid out in section 3.6

No.	Specification	Implemented
Cracking		
1	**Crack pattern should have a true three-dimensional representation**	Yes
2	**Fracture generator should be abstract from the object model**	Yes
3	**Crack pattern should depend upon an alpha texture**	Yes
4	**Crack pattern propagation should depend upon a specified material type**	Yes
5	**Simulate application of an area force instead of a point source** ***This desirable function was not implemented due to time constraints upon the project.***	No
6	**Simulate multiple synchronous and asynchronous impacts**	No

	This desirable function was not implemented due to time constraints upon the project.	
7	**Simulate minor deformation with a crack pattern** *This desirable function was not implemented due to time constraints upon the project.*	No
User Interaction		
8	**Choose the location(s) of impact point(s) on the object surface**	Yes
9	**User can manipulate their view of the world**	Yes
10	**A frame rate counter is displayed**	Yes
11	**User can specify the force applied during impact**	Yes
12	**User can adjust the speed of the cracking process and its animation** *The user can pause and continue the animation but the cracking process is performed in a single operation and its speed cannot be adjusted*	No
13	**User can specify an area for impact to occur** *The crack pattern itself covers a large area but the impact cannot be changed from being a point source*	No
Animation		
14	**Overall frame rate should not drop below 60fps**	Yes
15	**Overall frame rate should not drop below 100fps** *If a target object has > 1000 polygons cracking occurs in noticeable time, but this does not cause the frame rate to drop below 100fps*	Yes
16	**Animate the resulting objects with a physically based path** *This desirable function was not implemented due to time constraints upon the project.*	No
17	**Animate collision between objects changing their direction** *This optional function was not implemented due to time constraints upon the project.*	No
Object Modelling		
18	**All object should be represented by continuous polygon meshes** *The Fracture generator was unable to create a polygon mesh suitable for the cracking process. The resultant meshes from the cracking process were sealed polygon meshes but program errors caused them to be non-continuous.*	No
19	**Imported models should be saved in the .x (DirectX) file format** *Due to reasons given above this could not be implemented*	No
20	**Hollow object should be able to be imported and cracked**	Yes
Program Code		
21	**Program code should be highly efficient and bug-free**	Yes
22	**Program code should be well documented**	Yes

Table 5.1 - Specification fulfilment

Chapter 6: Evaluation

This chapter performs the evaluation laid out in section 3.7 upon the system implemented in chapter 5. However, due to the problems encountered during the implementation, documented in section 5.5, it was not feasible to perform a user evaluation of the fracture viewer. Without crack patterns generated from the Fracture generator, results of the cracking tests will not be at the level that the author would expect. Therefore, a visual evaluation of the cracking process against real-world shall be performed by the author.

6.1: System Overview

The flow diagram of the system in section 3.1 (figure 3.1) showed the manual and automatic stages of the project. As described in chapter 5, the implementation of the 3D crack pattern generation section was incomplete and therefore the crack patterns used in the fracture viewer had to be created manually. A revised flow diagram can be seen in figure 6.1 below.

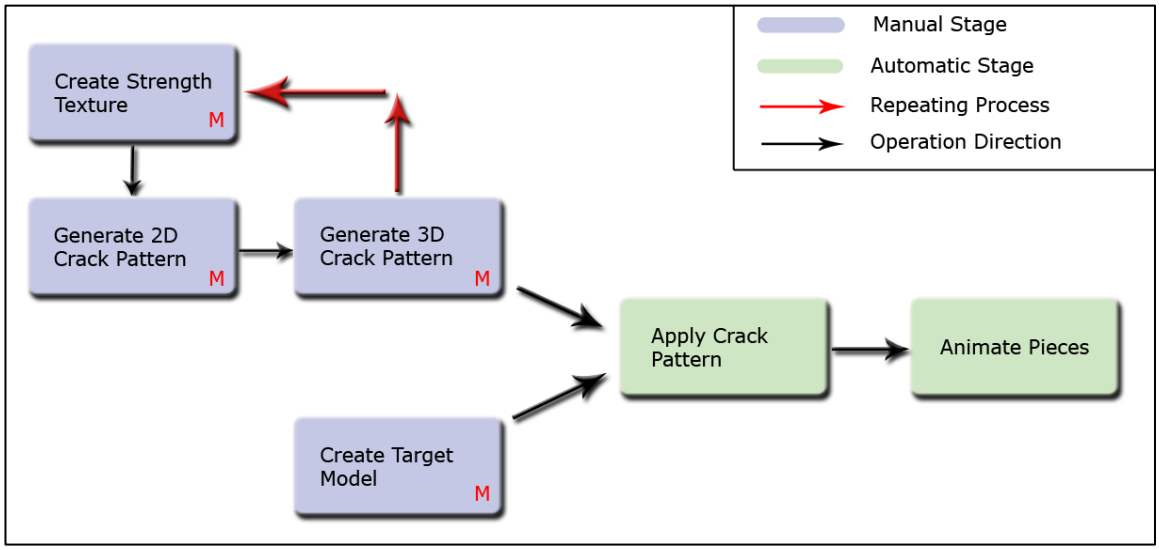


Figure 6.1 - Implemented system flow diagram. A red M indicates a manual stage

6.1: Crack Pattern Generation

The crack pattern generation produced results that the implementation was unable to convert into meshes to be applied in the fracture viewer; however, it was able to produce visual results that can be compared to real-world fractures. Taking crack patterns generated from the sample textures found in Appendix A the following results were obtained.

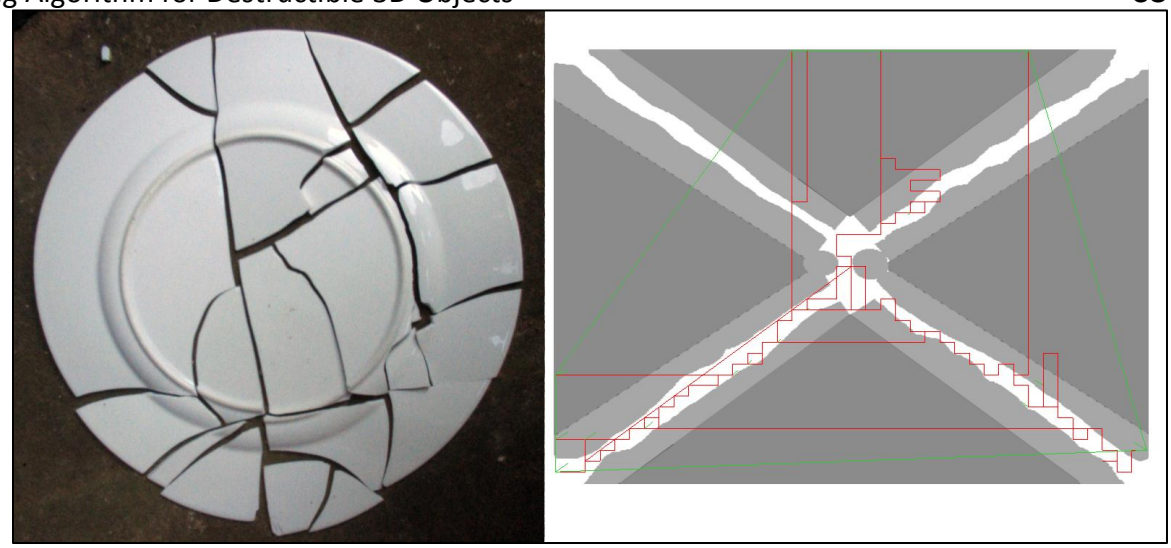


Figure 6.2 - A plate and a crack pattern generated from alpha 3

The crack pattern on the right of figure 6.2 was generated with alpha 3 which can be seen in the background of the crack pattern. The green lines show the border of the crack pattern created by the enclosing stage of generation. The crack pattern creates 15 separate volumes of varying sizes. The pattern follows a similar form to that seen in the real-world example on the left of figure 6.2. Although the crack pattern is not as detailed as one created by the fracture generator, it would produce an acceptable result when applied to an object in the fracture viewer.

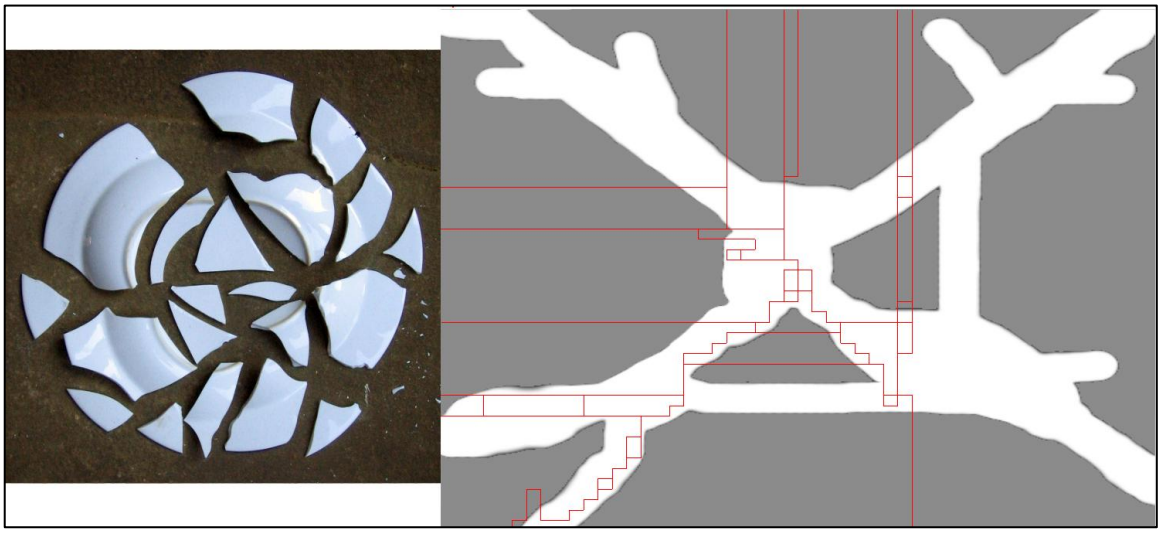


Figure 6.3 - A bowl and a crack pattern generated from alpha 6

Figure 6.3 shows a bowl dropped from 1 metre on to concrete and a crack pattern generated with alpha 6. It is possible that the crack pattern would produce such a fracture when applied in the fracture viewer; however, the real-world fracture occurred in a radial pattern whereas the crack pattern produced straighter branches with little variation in their direction. This is most likely a problem with the fracture generation algorithm itself. A more complicated algorithm with better path finding for the crack

branches would produce a more varied result and can produce crack patterns that closely resemble the real-world.

Figure 6.4 shows that some alpha textures are unsuitable for use with the fracture generator. With two single areas of solid colour the cracks head in straight lines until they reach the edge of the crack area. This produces very large volumes and very small volumes with very little in-between.

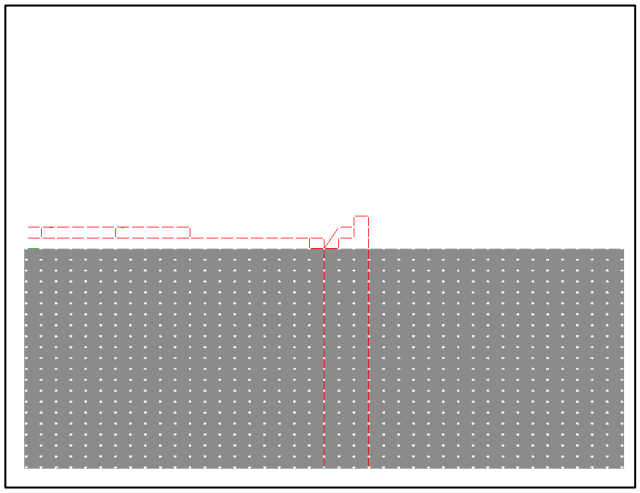


Figure 6.4 - A crack pattern created with alpha 1

Additional crack patterns created with each of the textures found in Appendix A along with 3D representations of the crack pattern can be found in Appendix D.

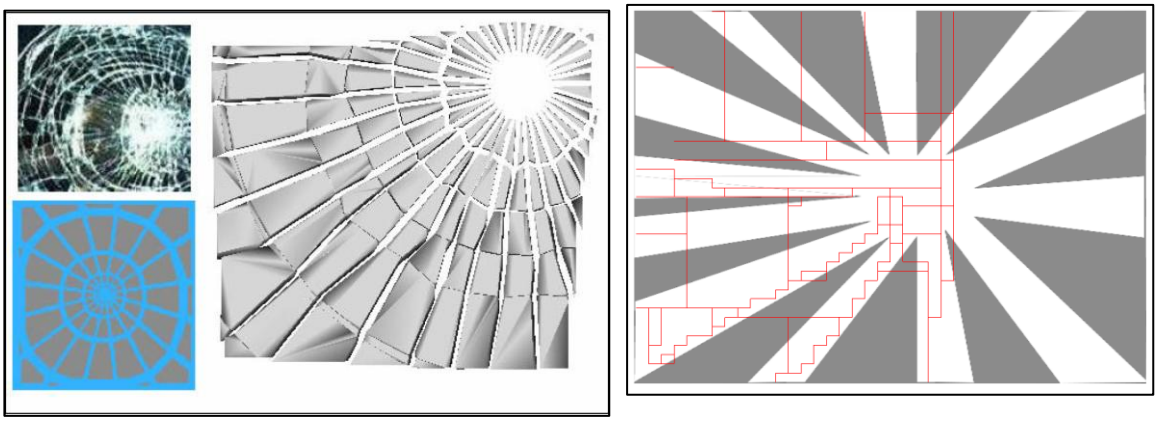


Figure 6.5 - From left to right) a comparison of the glass simulation produced by Fox in [2] and Miller in [3] against a crack pattern generated by alpha 4.

The crack pattern generated from alpha 4 (figure 6.5 right) attempted to reproduce the fracture commonly seen in shattered glass (figure 6.5 top left). However, comparison against crack patterns in [2] and [3] (figure 6.5 bottom left and centre respectively) which also attempt to reproduce the glass fracture do not produce good results. The radial pattern that Fox and Miller used in their projects produced a much more accurate crack pattern when compared against highly brittle materials such as glass. Whilst it may be possible to create a strength texture that produces an acceptable crack pattern for glass, the crack pattern generation algorithm would also need to be modified to either accept diagonal paths or have its branches follow the strength pattern closely.

6.2: Crack Pattern Application

Due to the problems during the implementation, a crack pattern was unable to be extracted from the Fracture generator. Therefore, evaluation of the fracture viewer was performed using two manually created crack patterns seen in figure 6.6

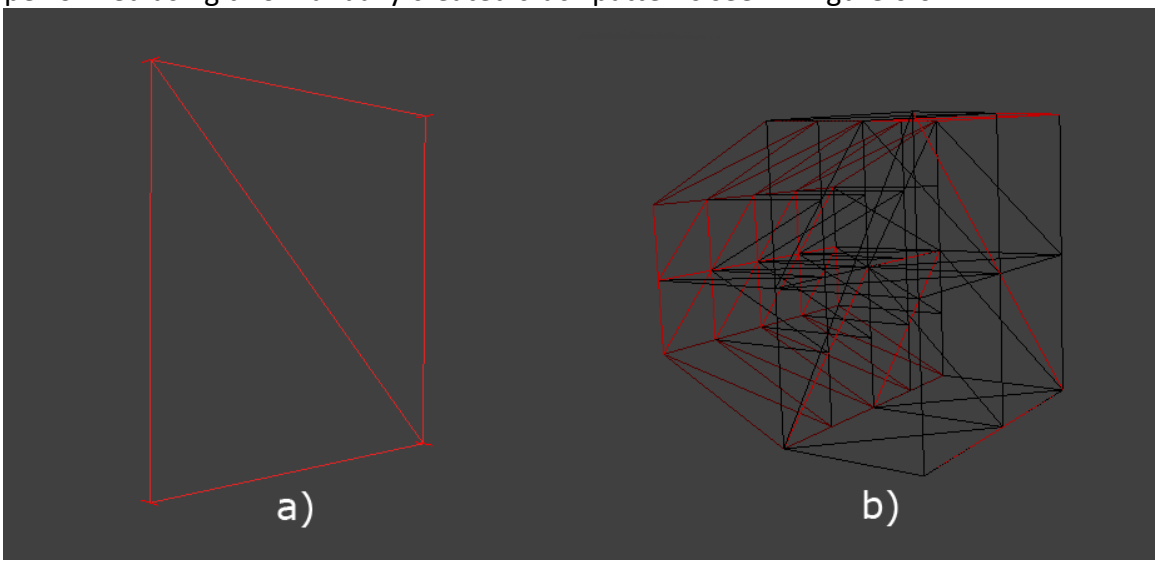


Figure 6.6 - Crack Patterns. a) is a double-sided plane. b) is a tapered box with 12 segments

Crack pattern a) is the simplest kind of crack pattern and was used solely for testing purposes during the creation of the fracture viewer. Results with a 1771 polygon model of a heart can be seen in figure 6.7.
The result shown in figure 6.7 indicates that simple cracking works correctly for models with curved polygon surfaces and correctly seals them afterwards.

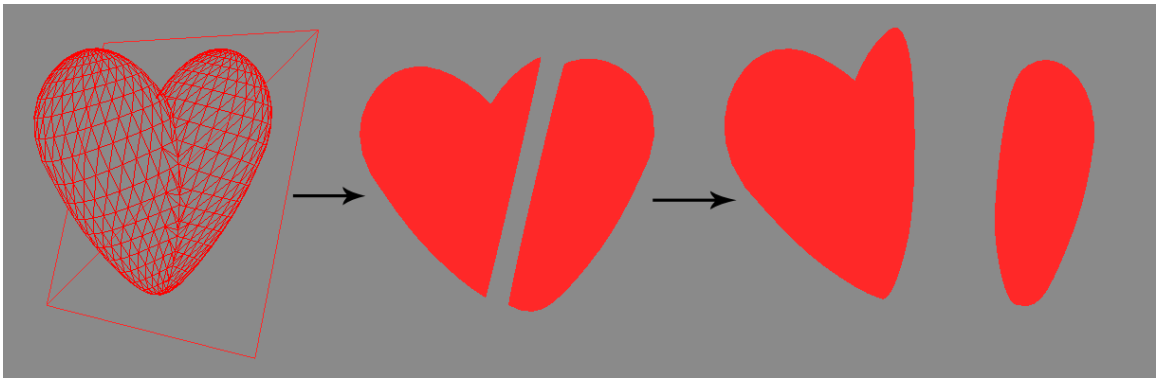


Figure 6.7 - Crack pattern application with a plane against a 1771 polygon heart model

6.2.1: Evaluation Against Real-World Fracture

For the following evaluation, three household items were tested; a small cup with handle, a shallow bowl and a dining room plate. 3D polygon models of each of these were obtained from [24] and tested in the fracture viewer with crack pattern b shown in figure 6.6.

Bowl

The bowl model was a 792 polygon model and was cracked with the impact point on the flat base of the bowl. The real-world bowl was dropped from a height of 1m on to a concrete surface. Figure 6.8 shows the results:

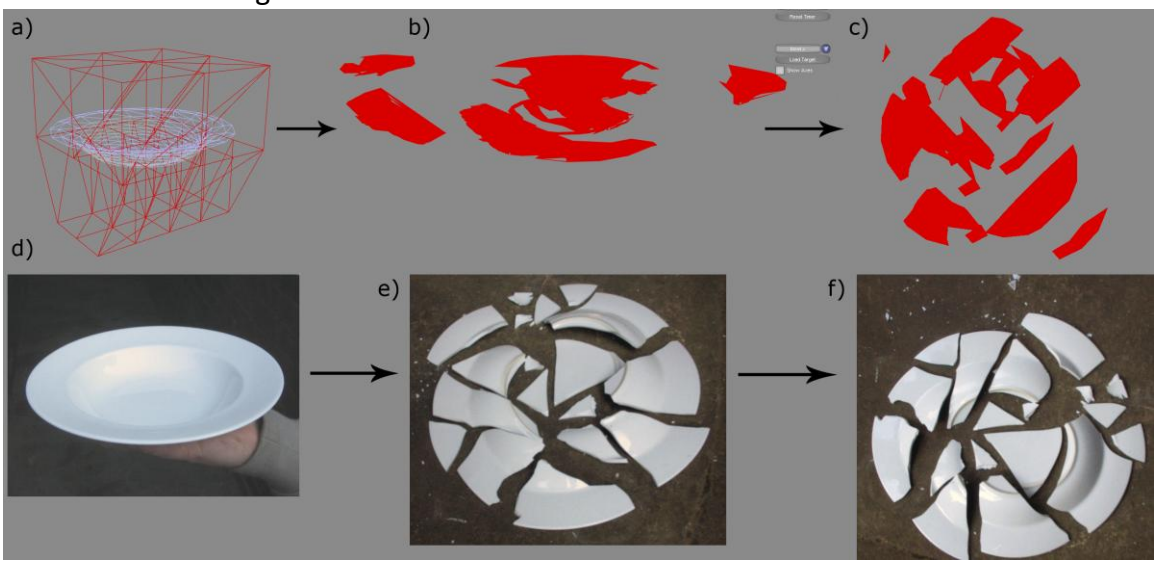


Figure 6.8 - Cracking a shallow bowl. a) shows the 3D bowl with its crack pattern in wireframe mode before cracking. b) shows a perspective view of the bowl after cracking. c) shows a bottom-up view of the cracked 3D bowl. d) shows the real-world bowl. e) shows the cracked bowl from a perspective view. f) shows the cracked bowl from a bottom-up view

The results in figure 6.8 compare exceptionally well to the real-world results. Figure 6.8 c) and f) show a very good similarity in shape and positioning of the pieces. However, a lack of smooth shading on the resultant pieces in figure 6.8 b) and c) take away from the accuracy of these pieces.

Cup

The small cup is a 3914 polygon model and was cracked from an impact point on the base of the model. The real-world cup was dropped on to concrete from a height of 1 metre. Figure 6.9 shows the results from this cracking process

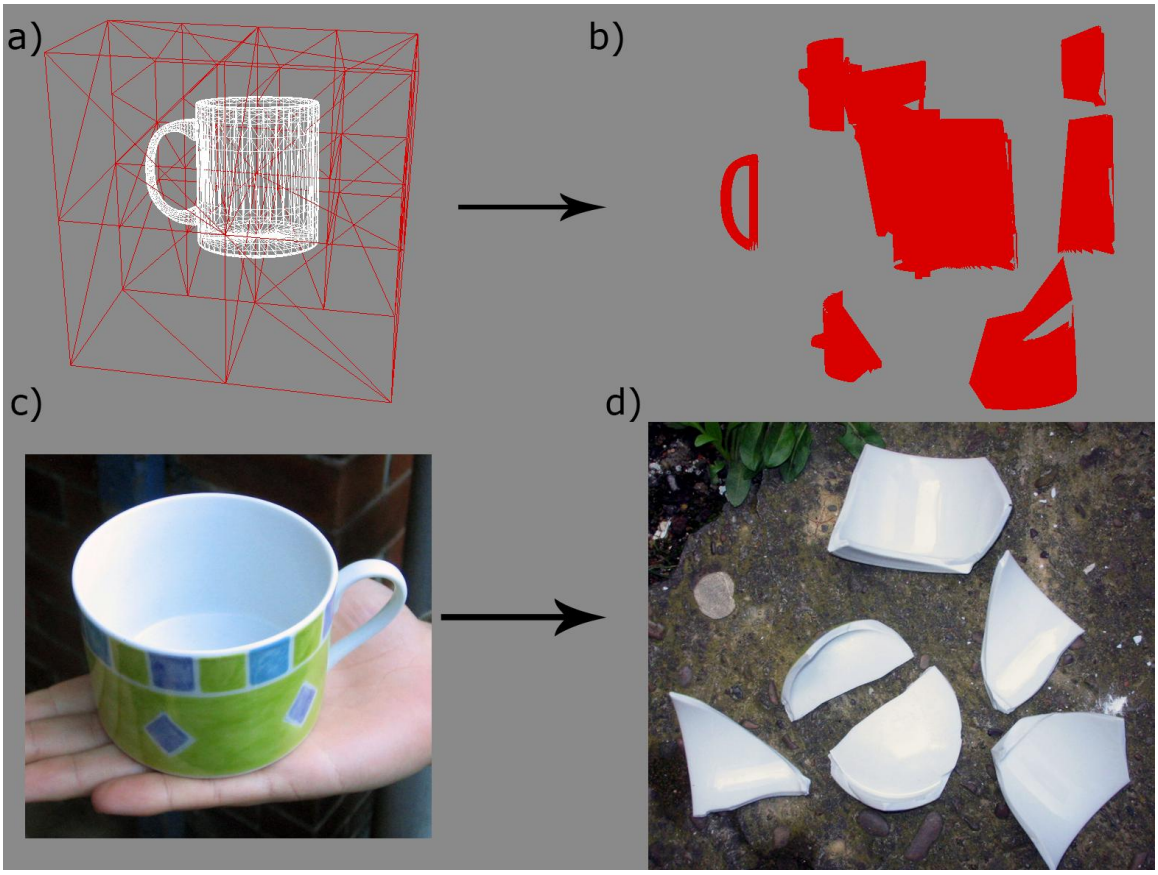


Figure 6.9 - Cracking a small cup. a) shows a 3D cup in wireframe mode contained by a crack pattern. b) shows the 3D representation of the cracked cup. c) shows a real-world shallow cup. d) shows the resultant pieces after cracking the cup (the handle could not be found)

The real-world cracking in figure 6.9 d) shows 6 large pieces formed from the breaking process. Figure 6.9 b) shows the same number of pieces with a similar disposition and shape. The resultant 3D pieces do not show the curved profile of the real-world fracture; this could be caused by the crack pattern as it was made with many straight lines at right-angles to each other instead of angular or curved volumes. The 3D pieces also demonstrate errors in the re-meshing algorithm as holes can be seen as well as jagged edges caused by the sealing algorithm. However, the results show an excellent result against the real-world results.

Plate

The plate is a 3360 polygon model and was cracked from an impact point at an angle to the edge of the plate. The real-world plate was also dropped to an impact point at a similar angle from a distance of 1 metre above a concrete floor. Figure 6.10 shows the results of this process:

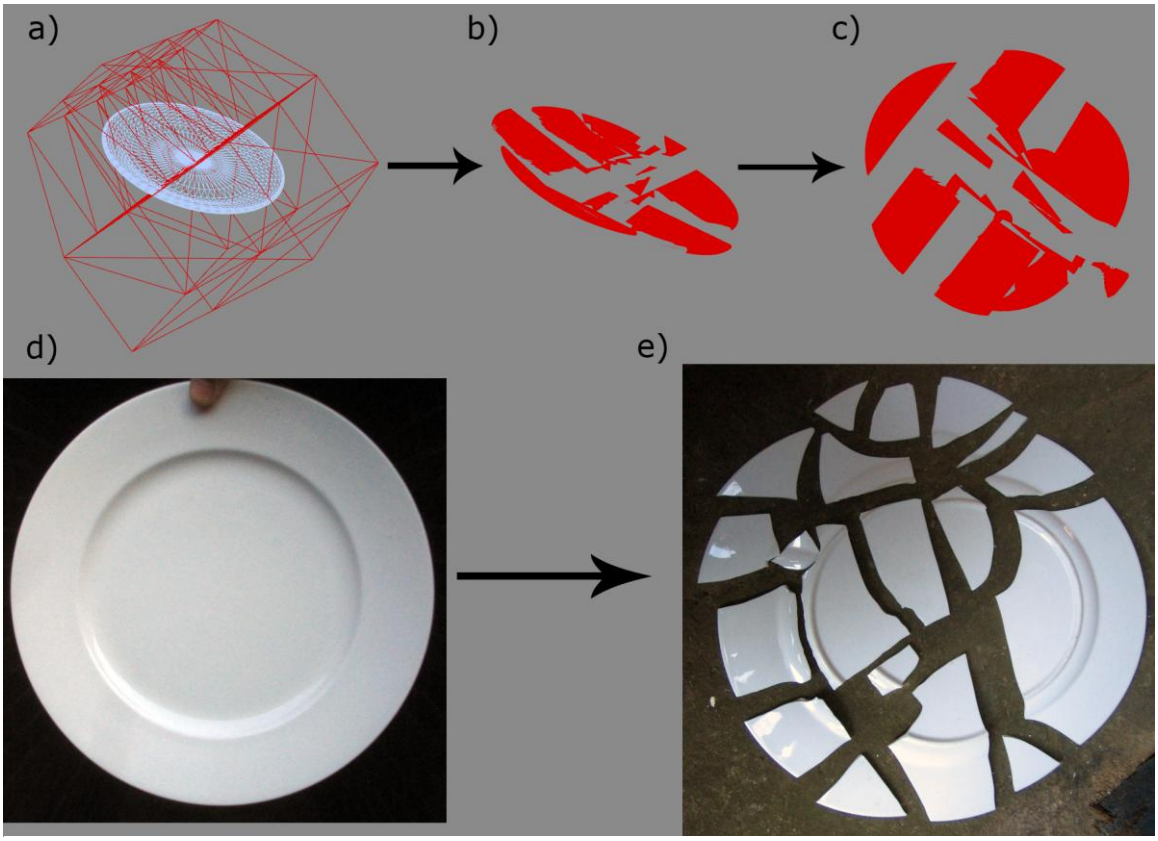


Figure 6.9 - Cracking a plate. a) shows the 3D plate in wireframe mode surrounded by the crack pattern. b) shows the cracked 3D plate from a perspective view. c) shows the cracked plate from a bottom-up view. d) show an un-cracked real-world plate. e) shows the cracked plate from a bottom-up view

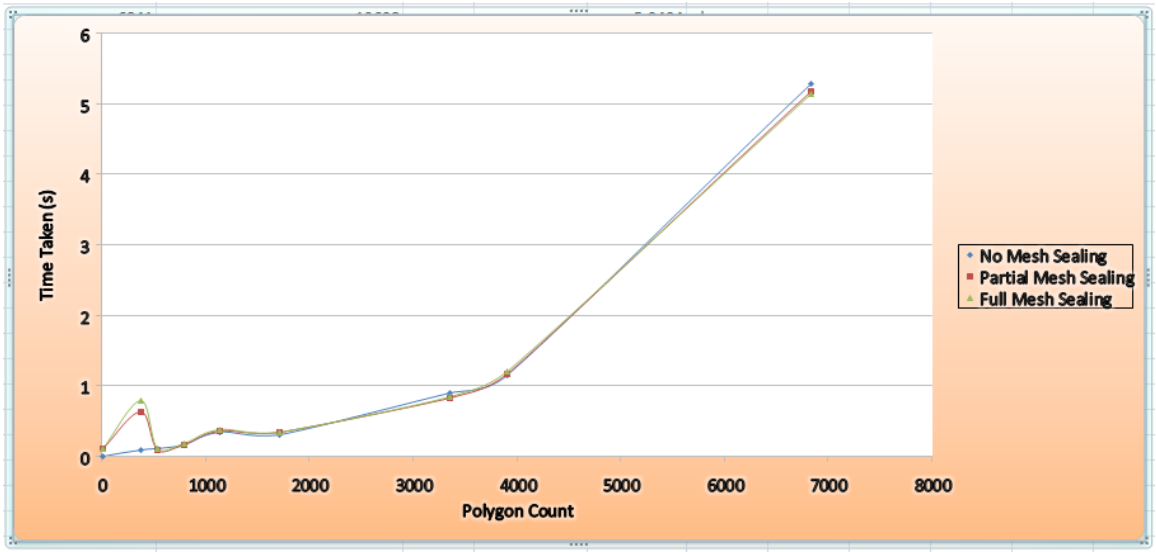
The fracture shown in figure 6.10 shows good results for the given crack pattern. In a similar vein to the tests with the bowl and the cup the crack pattern had produced a number of large pieces with the majority of these having straight edges. This is unfortunate as a better crack pattern would not result in the 3D pieces having the straight edges. A number of smaller pieces were also produced from the plate. The result compares well to the real-world example shown in figure 6.10 e).

6.3: Performance Testing

The performance of the two systems is very important as real-time cracking is one of the primary aims of this project. It was stated in chapter 4 that the fracture generator was an offline process and therefore performance was not an issue. For this reason, performance of the fracture generation process will not be evaluated.

The performance of the fracture viewer was evaluated by measuring the time taken to complete the cracking process. This was done using a high-precision timer available through the Win32 libraries accessible in C#. The cracking process was tested with 6 different models with increasing numbers of polygons. Each test was performed 3 times

with the crack pattern aligned upon a different axis each time. Each test was performed without mesh sealing, with partial mesh sealing and with full mesh sealing. An average of these results was taken and a graph of time taken against polygon count can be seen in figure 6.11.



Polygon Count	12	384	540	792	1140	1717	3360	3914	6841
No Mesh Sealing	0.0073	0.0846	0.1034	0.16845	0.3341	0.3091	0.89319	1.15687	5.2821
Partial Mesh Sealing	0.10693	0.6308	0.0986	0.16963	0.3643	0.3386	0.83523	1.16625	5.1675
Full Mesh Sealing	0.1143	0.7845	0.1158	0.17465	0.3715	0.3376	0.84551	1.20135	5.1371

Figure 6.10 - Performance Graph for Fracture Viewer, Polygon Count against Time Taken

Figure 6.11 shows an increasing trend in time taken with larger polygon counts. This trend increases rapidly after 3000 polygons. The general trend of the graph indicates that polygon cracking is an $O(n^2)$ process. It should be noted that the anomaly with the 384 polygon model where sealing the mesh lead to a 6-fold increase in time taken can be attributed to the model itself. Meshes with large thin planes can be very difficult to seal and therefore take more time. Models such as the 1140 polygon model caused a small anomaly in the trend as the model was not properly centred about the origin and had a large concentration of polygons in one portion of the model. This uneven weighting caused the small increase in time taken.
It should be noted that other than the above anomalies there was very little difference in time taken whether the mesh was sealed or not. At times when the unsealed mesh took more time to crack than a sealed version can be attributed to small variations in the position and angle of impact causing a different crack to occur.

6.4: Problems and Errors

Closer inspection of the meshes that resulted from the cracking of each of the objects revealed problems with the re-construction methods used in the fracture viewer.

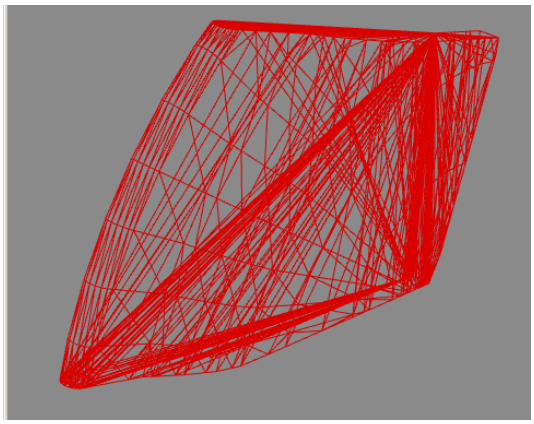


Figure 6.11 - Re-meshing errors

Figure 6.12 demonstrates an error with the sealing process for the resultant meshes. Large concentrations of adjacent lines lead to very small polygons which cause large polygon counts in the resultant meshes. This leads to a drop in the frame-rate of the system once the mesh has been cracked. In a computer game situation where many of these cracks could happen at the same time, the unnecessarily increased polygon count will cause problems for the performance of the game.

Figure 6.13 shows one of the resultant pieces after cracking the plate mesh. This mesh appears to have been outside of the influence of the crack pattern as it has not been sealed and has only had polygons removed from it. This behaviour is interesting as this particular piece is along the central axis of the original model. This seems to be a problem with the manual creation of the crack pattern. It seems most likely that there is a small gap between two of the volumes of the crack pattern which caused this area to miss being cracked.

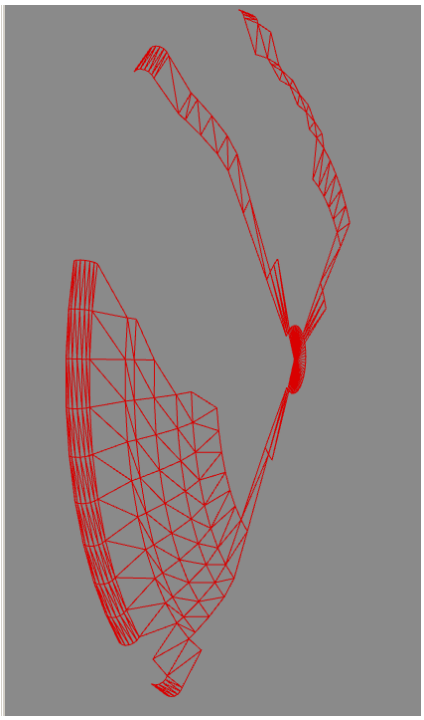


Figure 6.12 - Crack pattern errors

6.5: Future Work

6.5.1: Complete Offline Crack Pattern Generation

Due to time constraints in this project an automatic method of generating crack pattern meshes could not be completed. The method of generating crack pattern meshes against alpha textures has been found to be successful; however, a successful technique to find volumes in the crack pattern could not be implemented. Theoretically, both the volume sweeping and volume tracing methods should correctly identify the volumes in the crack pattern; however, errors in the paths of the crack patterns caused infinite loops to occur. Creating more accurate crack patterns would relieve this problem and allow the volumes to be found. An automatic method for generating crack patterns would make the task of creating a crack pattern library for the fracture viewer process far simpler and free of any human errors which lead to the error seen in figure 6.13.

6.5.2: Removal of Clipping and Mesh Sealing Errors

As identified in section 6.4, two principal errors appear when constructing the resultant pieces from a fracture. Firstly, the final case for clipping described in section 5.3 as Clip3 has not yet been implemented. This causes inaccuracies in the resultant meshes which could cause the mesh to have jagged edges or holes in the mesh. A simple solution to this would be to implement a Clip3 algorithm.
Mesh sealing errors as seen in figure 6.12 cause additional polygons to be drawn and over hollow objects i.e. the cup, can cause the mesh to over supposedly open areas. For this, a more advanced method of sealing i.e. Delaunay triangulation could be used. These seals should also be able to occur against multiple planes as the clipping does. This is not implemented into this project but it a candidate for further investigation.

6.5.3: Adding Detail with Displacement Maps

A simple crack pattern like that seen in figure 6.5 relies on the complexity of the polygon model and the angle of impact to create the illusion of cracking. However, crack patterns like this require less time to crack the object (as the cracking process is repeated for each volume and tested against every polygon in the crack pattern, a crack pattern with more polygons will take longer to complete the operation). To resemble the complexity of the crack pattern, a displacement map could be applied to any area of the mesh which has been sealed (i.e. where the crack pattern has split the model). This displacement map could be an addition to the crack pattern library for each of the crack patterns. Potentially, normal or bump maps could be used instead of displacement maps so that the clipped surface does not have to contain additional polygons to produce the same ‘rough’ effect. This would allow for simpler crack patterns to be used which would make the crack pattern mesh generation easier.

6.5.4: Physics Based Animation and Collision Detection

One of the principal aims of this project was to make the cracking process as realistic as possible. This would have been aided by projectile based animation of the mesh’s resultant pieces and collision detection as a post-crack process. This will produce better demonstrations for the process so that the fractures could be evaluated by users of the system. However, with the advent of modern computer games and physics engines, if this system was integrated into a modern game engine then this post-process would be handled by the game engine and not the cracking process. Therefore, although it would be useful for evaluation testing, physically based animation and collision detection is not an essential part of the system; however, it is still a candidate for further work.

Chapter 7: Conclusions

This project set out to create a method of realistically destroying or fracturing 3D polygonal objects in real-time. This technique was targeted for use in computer games to produce better interactive environments for the latest games.

Research for this system came from variety of sources. The project was most heavily influenced by Fox [2] and Miller [3] yet the base for this entire project was the work by O'Brien [10]. Additional influence came from attempts to simulate this in commercial computer games such as Red Faction [13].

It became clear early on that in order to maintain the real-time aspect of this system there would have to be an offline process to generate crack patterns. The Fracture generator created patterns in a way that could be directly influenced by an artist with the use of an alpha texture. This gave the artist much more control over how the crack pattern would look so that objects in the real-time environment would fracture in different ways dependent upon their material strength set out by the artist. However, due to difficulty with the implementation of the volumization algorithm, this module of the system was unable to output crack patterns in the desired format. If further work was to be carried out in this area, the modular design of the real-time and offline systems means that the real-time program does not have to be rebuilt as it will accept crack patterns created with any generator.

With a manually created crack pattern, the real-time aspect of the system, the fracture viewer, was able to successfully crack objects in to the volumes set out in the crack pattern. The performance of this system was good; however, the time to complete the cracking process was noticeable in models with more than 1000 polygons. Using this technique on low-polygon objects such as walls, glass and small to medium sized household objects would be an excellent way to simulate a dynamic environment. That said, improvements to the cracking algorithm could be made. Using a high-performance programming language i.e. C++, would improve the speed of the algorithm so that this polygon limit may be increased; however, the choice of C# as the programming language was the best language for the rapid development of the project at the time. An optimized sealing algorithm would decrease the number of polygons to be rendered in the resultant pieces, thereby increasing rendering speeds. The use of a bounding sphere test with each volume in the crack pattern could also increase performance. With the advent of multiple processor desktop computers this process could be threaded and performed in parallel so that the performance of this process is increased significantly.

The quality of the resultant pieces was generally good. As described in section 6.4, problems were encountered re-meshing the pieces. Whilst this caused problems with more complex crack patterns, figure 6.7 showed that the re-meshing process could work

successfully. The quality of the sealing algorithm was acceptable but further work into this area of the algorithm should be performed.

If this system was added to a physically based environment (by the inclusion of a physics engine in the computer game) the resultant pieces of a cracked mesh would be animated in a realistic manner. This would make the destruction of each 3D object look realistic and therefore suitable for a commercial product.

References

[1]: Moore, G.E, Electronics Magazine, 19 April 1965

[2]: Fox, Daniel (2003), Demolishing Objects in Computer Games; 3rd Year Project, University of Sheffield

[3]: Miller, Adam (2004), A Cracking Algorithm for Exploding Objects; 3rd Year Project, University of Sheffield

[4]: http://en.wikipedia.org/wiki/Image:Sodium_chloride_crystal.png; Walker, Martin (2005)

[5]: http://en.wikipedia.org/wiki/Image:Stress-strain1.png; Wikipedia user: Moondoggy (2005)

[6]. Anderson T.L. (1995), Fracture Mechanics: Fundamentals and Applications Second Edition, CRC Press LLC.

[7]: Jerry Ballard of Virginia Tech Materials Science and Engineering http://www.sv.vt.edu/classes/MSE2094_NoteBook/97ClassProj/exper/ballard/www/ballard.html (Last visited 20/11/2005)

[8]: Terzopoulos, D and Fleischer, K (1988), Deformable Models. *The Visual Computer,* 4:306-311

[9]: Terzopoulos, D and Fleischer, K (1988), Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture. *SIGGRAPH '88 Proceedings* vol. 22, pg 269-278

[10]: O'Brien, J. F., Hodgins, J. K., (1999), Graphical Modeling and Animation of Brittle Fracture, *The proceedings of ACM SIGGRAPH 99*, Los Angeles, California, August 8-13, pp 137-146.

[11]: Schöberl, J. NETGEN—An advancing front 2D/3D-mesh generator based on abstract rules. *Comput. Visualiz. Sci.* 1 (1997), 41–52; see also www.sfb013.uni-linz.ac.at/~joachim/netgen/

[12]: Müller, M., McMillan, L., Dorsey, J., Jagnow, R. (2001), Real-Time Simulation of Deformation and Fracture of Stiff Materials, (Eurographics CAS), Computer Animation and Simulation 2001, Springer-Verlag Wien, 113-124

[13]: http://redfaction.volitionwatch.com/faq/geomodfaq.shtml

[14]: Watt, Alan (2000), 3D Computer Graphics 3rd Edition

[15]: http://www.unrealengine.com – Images created using Unreal Engine 3 by Epic Games

[16]: Miller, Tom (2003), Kick Start Managed DirectX 9 Graphics and Game Programming

[17]: http://www.programmersheaven.com/2/FAQ-DIRECTX-What-Is-Mipmapping (Last visited 24/11/2005)

[18]: http://msdn.microsoft.com/vcsharp/ (Last visited 02/04/2006)

[19]: http://java.sun.com (Last visited 02/04/2006)

[20]: http://www.opengl.org (Last visited 02/04/2006)

[21]: http://msdn.microsoft.com/directx (Last visited 02/04/2006)

[22]: http://www.whisqu.se/per/docs/math13.htm - Narkhede, A & Manocha, D, Fast Polygon Triangulation based on Seidel's Algorithm (Last visited 5/4/2006)

[23]: http://www.cs.berkeley.edu/~jrs/papers/2dj.pdf - Shewchuk, J.R (2001), Delaunay Refinement Algorithms for Triangular Mesh Generation

[24]: http://www.turbosquid.com (Last visited 26/04/2006)

[25]: http://www.codeproject.com/csharp/cspolygontriangulation.asp (Last visited 26/04/2006)

[26]: http://msdn.microsoft.com/vstudio/ (Last visited 26/04/2006)

[27]: http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112 (Last visited 26/04/2006)

[28]: http://www.swissquake.ch/chumbalum-soft/ (Last visited 26/04/2006)

Appendix A: Test Textures

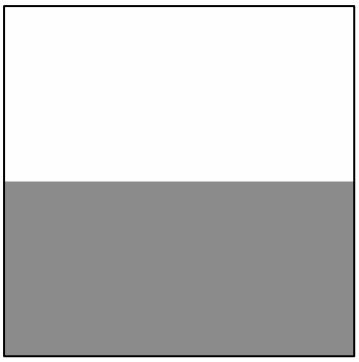


Figure A.1 - Alpha Texture 1

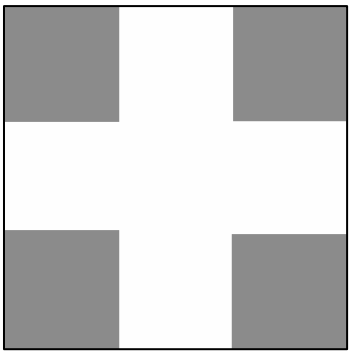


Figure A.2 - Alpha Texture 2

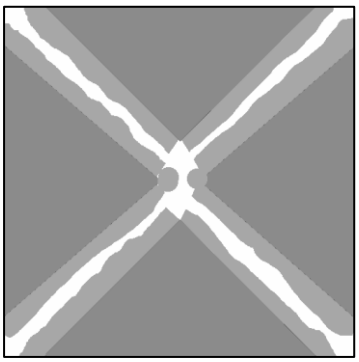


Figure A.3 - Alpha Texture 3

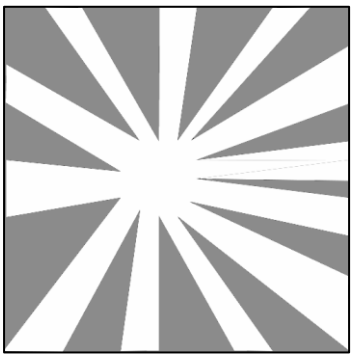


Figure A.4 - Alpha Texture 4

Figure A.5 - Alpha Texture 5

Figure A.6 - Alpha Texture 6

Appendix B: Kinematics

Kinematics is the branch of mechanics that studies the motion of a body or a system of bodies without consideration given to its mass or the forces acting on it. There are 5 primary kinematic equations involving distance from origin (s), initial velocity (u), current velocity (v), acceleration (a) and time (t).

$$S = ut + \tfrac{1}{2}at^2$$

Equation B.1 - Distance

$$V^2 = u^2 + 2as$$

Equation B.2 - Current velocity squared

$$V = u + at$$

Equation B.3 - Current Velocity

$$S = vt - \tfrac{1}{2}at^2$$

Equation B.4 - Distance relative to current velocity

$$S = ((v - u) / 2)t$$

Equation B.5 - Distance not requiring acceleration

These when supplied with an initial velocity and acceleration (generally $9.81ms^{-2}$, the acceleration caused by gravity) the arc of a projectile can be calculated against time. Implementing these equations in to the explicit animation of the object shards will provide a realistic motion of the object's fragments therefore improving the realism of the simulation.

Appendix C: Clipping Methods

As stated in chapter 5, 3 clipping methods were implemented in to the cracking algorithm. These clipping methods, described as Clip1 and Clip2 relate to the number of vertices that were accepted by a volume. The re-meshing method for each of these is described below.

Clip1

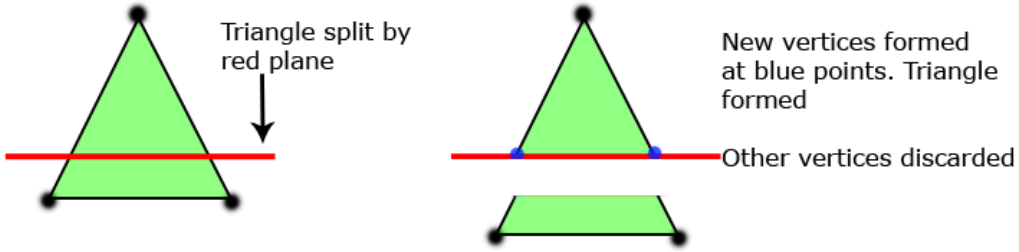


Figure C.1 - Clip1

Clip2

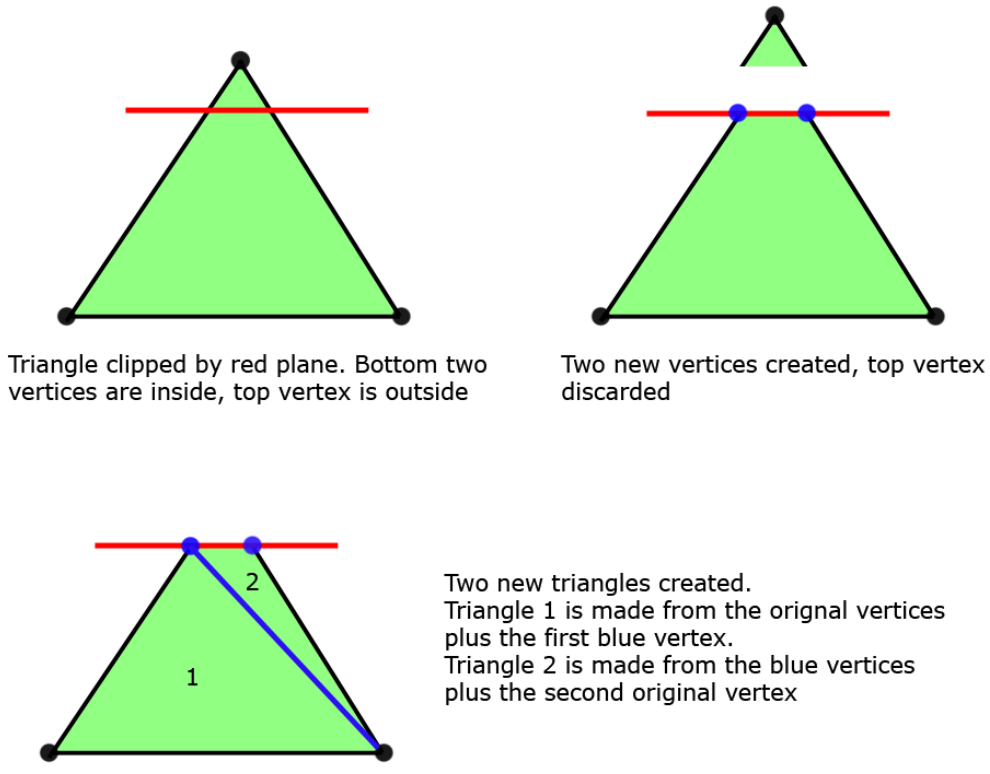


Figure C.2 - Clip2

Appendix D: Fracture Generator Examples

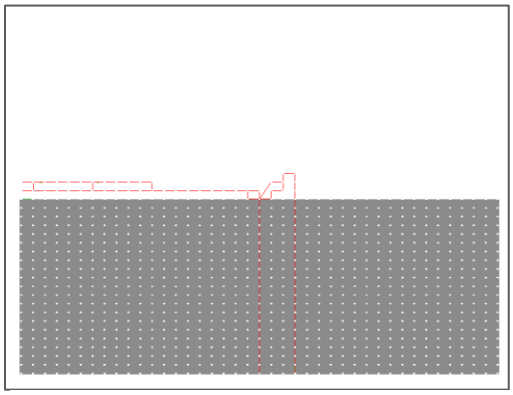


Figure D.1 - Alpha 1

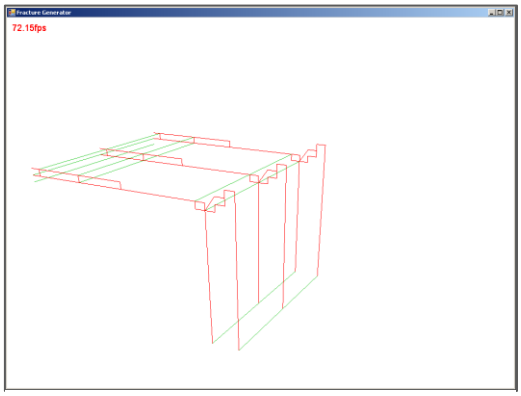


Figure D.2 - Alpha 1 3D perspective view

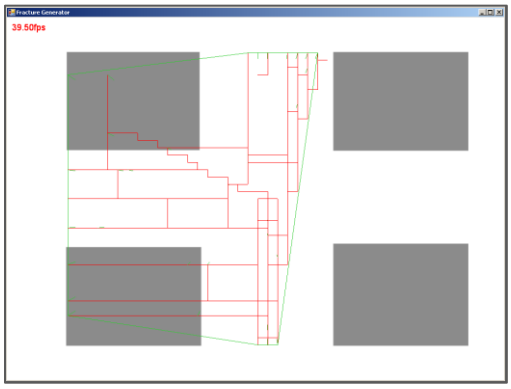


Figure D.3 - Alpha 2

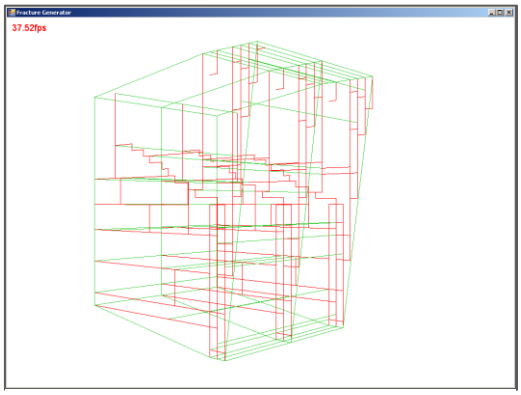


Figure D.4 - Alpha 2 3D perspective view

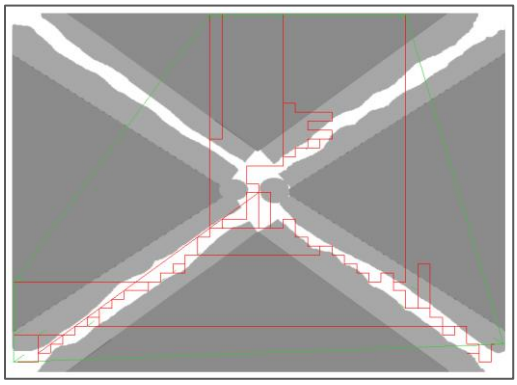


Figure D.5 - Alpha 3

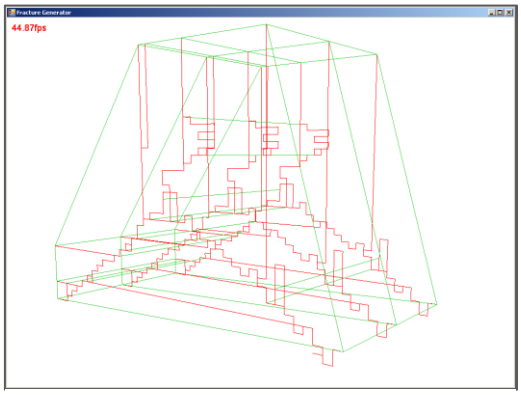


Figure D.6 - Alpha 3 3D perspective view

Figure C.2 – Clip2

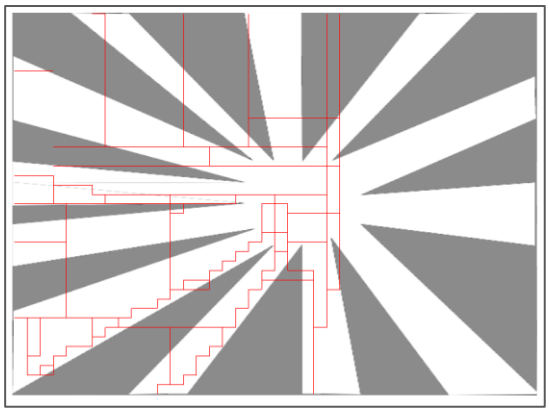


Figure D.7 - Alpha 4

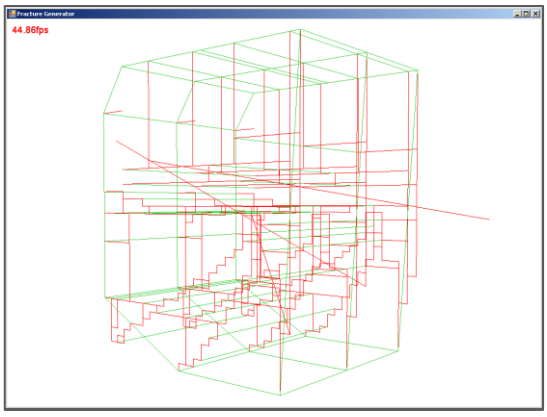


Figure D.8 - Alpha 4 3D perspective view

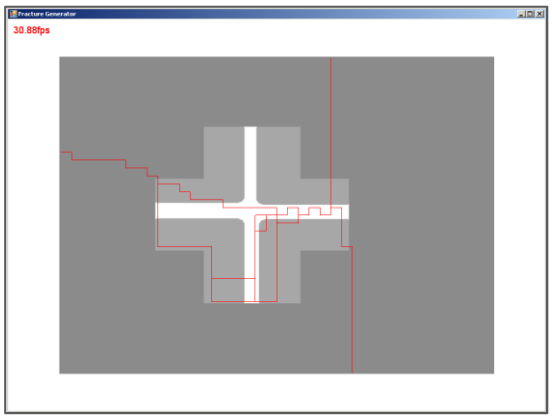


Figure D.9 - Alpha 5

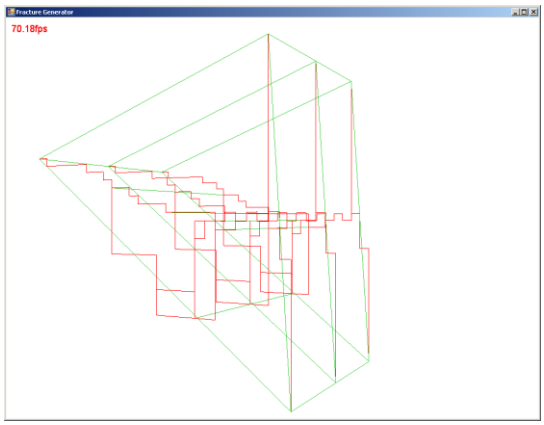


Figure D.10 - Alpha 5 3D perspective view

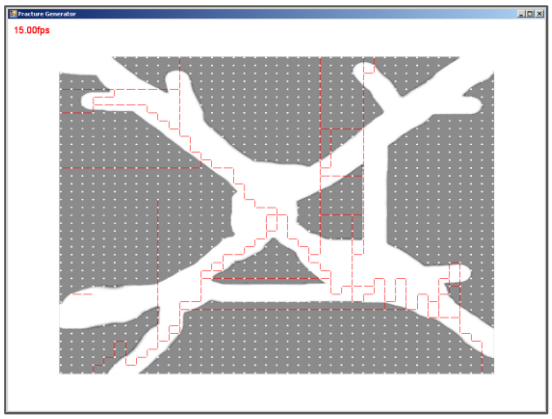


Figure D.11 - Alpha 6

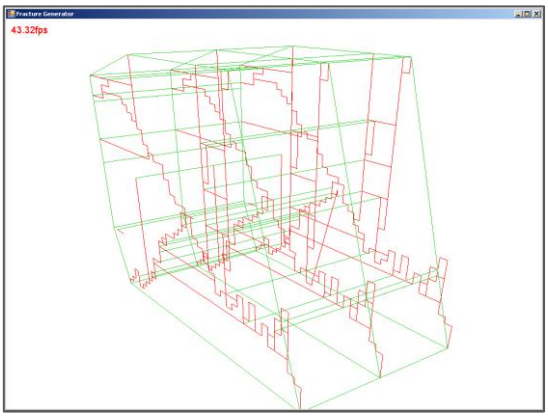


Figure D.12 - Alpha 6 3D perspective view