Real-time procedural modeling of fracture of brittle materials in games

Sheikh Dawood Abdul Ajees

2009

Sheikh Dawood Abdul Ajees. (2009). Real-time procedural modeling of fracture of brittle materials in games. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/14962>

<https://doi.org/10.32657/10356/14962>

NANYANG TECHNOLOGICAL UNIVERSITY

REAL-TIME PROCEDURAL MODELING OF FRACTURE OF BRITTLE MATERIALS IN GAMES

SHEIKH DAWOOD BIN ABDUL AJEES

School of Computer Engineering

A thesis submitted to the Nanyang Technological University

in fulfilment of the requirement for the degree of

Master of Engineering

2009

Acknowledgements

I would like to take this opportunity to express my most sincere gratitude to TQ Global Singapore and Nanyang Technological University for providing me the chance to engage in this research project.

Firstly, I would like to thank my project supervisor, Assistant Professor Henry Johan, from the School of Computer Engineering, for his invaluable advice and constant guidance during the term of the project. Throughout the project, he has given me interesting ideas to work on. His interest in this project has given me a passion in game development.

I would also like to extend my thanks to the staff of TQ Global who have given me creative ideas for my project, and provided me with all the resources I need.

Last but not least, I would like to thank all my colleagues who have given me constructive comments on my work and assist me in solving problems.

Contents

Acknowledgements	ii
Contents	iii
List of Tables.....	vi
List of Figures	vii
Summary	ix
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Scope.....	2
1.3 Assumptions	2
1.4 Challenges.....	3
1.5 Methodology	5
1.6 Our Contributions.....	6
1.7 Overview.....	7
1.8 Summary	7
Chapter 2 Background.....	8
2.1 Fracture Mechanics	8
2.1.1 Dynamic Fracture Mechanics	8
2.1.2 Brittle Fracture	9
2.1.3 Strength of Materials.....	9
2.2 Crack Generation and Propagation	10
2.3 Physics in Games.....	14
2.3.1 Rigid Body Dynamics.....	15
2.3.2 Collision Detection and Collision Response	16
2.4 Summary	19
Chapter 3 Related Work.....	20
3.1 Physically-based Modeling	20
3.1.1 Finite Element Model.....	20
3.1.2 Spring-mass Model.....	23
3.1.3 Point-mass Model.....	23
3.1.4 Explosion Model	24

3.1.5	Static Crack Patterns	25
3.2	Meshless Model.....	26
3.3	Procedural Modeling.....	26
3.4	Summary	29
Chapter 4	Fracture Framework	31
4.1	Fracture Framework Overview	31
4.2	Breakable Material.....	33
4.2.1	Material Settings.....	33
4.2.2	Strength Threshold	33
4.2.3	Probability Texture.....	35
4.2.4	Shape Mask.....	38
4.3	Fracture Initiation	40
4.4	2D Crack Propagation	41
4.4.1	2D Crack Pattern Test Program	41
4.4.2	Data structure	42
4.4.3	Getting boundary mask of object.....	43
4.4.4	Crack propagation	45
4.5	3D Mesh Generation.....	56
4.5.1	Get boundary vertices of fragments and number of fragments	56
4.5.2	Triangulate boundary vertices	58
4.5.3	3D extrusion and wrap-around vertices	59
4.5.4	Physics properties	60
4.6	Summary	61
Chapter 5	Results.....	62
5.1	Comparisons.....	62
5.1.1	16-way and 8-way crack propagation comparison	62
5.1.2	Pixel resolution of probability texture comparison	64
5.2	Visual Results	67
5.2.1	Destruction of walls.....	67
5.2.2	Destruction of arbitrary shaped objects	79
5.2.3	House destruction.....	82
5.2.4	Application in a game environment.....	85
5.3	Performance	86
5.4	Summary	89
Chapter 6	Conclusions	90
6.1	Summary	90

6.2 Future Work 91

Bibliography 93

List of Tables

Table 2.1	Strength table of specific materials.	10
Table 3.1	Comparison of current methods.	29
Table 4.1	Strength threshold of material.	34
Table 5.1	Comparison of features between the 16-way and the 8-way methods.	64
Table 5.2	Specification of test machine	86
Table 5.3	Performance table for an object with a texture size of 64x64 pixels.	87
Table 5.4	Performance table for a wall broken with varying texture sizes.	88

List of Figures

Figure 1.1	Examples of a concrete wall being dynamically fractured.....	2
Figure 1.2	Example of objects with flat extruded cross-sectional surface.....	3
Figure 2.1	Cracks on brick walls.	11
Figure 2.2	Cracks on concrete.	12
Figure 2.3	Cracks in wood.....	13
Figure 2.4	Cracks on glass.....	13
Figure 2.5	Types of bounding volumes.	17
Figure 3.1	A tetrahedralized mesh.	21
Figure 3.2	An eagle represented as a cube mesh.	22
Figure 3.3	Destruction of a wall with a finite element model.	22
Figure 3.4	Results of a shattered glass with a point-mass model.	24
Figure 3.5	Results from procedurally modeling of fracture.....	27
Figure 4.1	Flow diagram of fracture framework.....	32
Figure 4.2	Material settings file.	33
Figure 4.3	Photo of grain lines on wood and probability textures of wood.	36
Figure 4.4	Photo of cracks on glass and probability textures of glass.	36
Figure 4.5	Photo of cracks on concrete and probability textures of concrete.....	36
Figure 4.6	A probability texture with noise.	37
Figure 4.7	Custom probability textures.....	38
Figure 4.8	Shape masks of different objects such a heart-shape and bunny.	39
Figure 4.9	Creating shape masks from 3DS Max.	39
Figure 4.10	Mapping of probability texture to object.	40
Figure 4.11	Screenshot of crack propagation test program.	41
Figure 4.12	Graph representation of the crack pattern.	42
Figure 4.13	Bitmap representation of the crack pattern.	43
Figure 4.14	Neighbouring pixel for simple edge detection.	44
Figure 4.15	Neighbouring 8 pixels for boundary mask traversal.	45
Figure 4.16	Neighbouring pixels for crack propagation.....	46
Figure 4.17	Pixel probability pseudo code for first degree neighbour.	46
Figure 4.18	Pixel probability pseudo code for second degree neighbour.	47
Figure 4.19	Angle between previous and next neighbour.	48
Figure 4.20	A normal cosine function.....	49

Figure 4.21	Cosine function of Equation 12.	50
Figure 4.22	Angle between start of branch and next neighbour.....	50
Figure 4.23	Cosine function of Equation 13.	51
Figure 4.24	Selection of first neighbour to propagate crack.	53
Figure 4.25	Selection of second neighbour to propagate crack.	54
Figure 4.26	Selection of third neighbour to propagate crack.....	54
Figure 4.27	Graph with terminating nodes.	57
Figure 4.28	Graph traversal for boundary vertices.....	58
Figure 4.29	A simple polygon triangulation.	59
Figure 4.30	An extruded polygon.	60
Figure 5.1	Comparison of 16-way (left) and 8-way (right) crack propagation.	63
Figure 5.2	Crack patterns generated with an “S”-shaped probability texture.....	63
Figure 5.3	Probability and mask texture for resolution test.	65
Figure 5.4	Crack pattern graph using resolution 16 (left) and 32 (right).....	65
Figure 5.5	Crack patterns using higher resolution textures.....	66
Figure 5.6	Probability textures of glass, concrete and wood.....	68
Figure 5.7	Photo of shattered glass.	68
Figure 5.8	Shattering a glass wall.	70
Figure 5.9	Breaking a concrete wall.	71
Figure 5.10	Breaking of a wooden wall.	72
Figure 5.11	A probability texture with an “S”.	73
Figure 5.12	The “S”-shaped crack pattern on a broken wall.	74
Figure 5.13	Probability textures with a hexagonal and square grid.....	74
Figure 5.14	Broken wall with noise probability texture.	75
Figure 5.15	Wall with an “S”-shaped crack pattern.	76
Figure 5.16	Broken wall with a hexagonal crack pattern.....	77
Figure 5.17	Broken wall with a square grid crack pattern.	78
Figure 5.18	Shape mask of a hand (left) and heart (right).	79
Figure 5.19	A broken hand.....	80
Figure 5.20	A broken heart.....	81
Figure 5.21	Model of a house.....	82
Figure 5.22	Shape mask of the walls of the house.	83
Figure 5.23	Destruction of a house.	84
Figure 5.24	Screenshots in a game.	85

Summary

This thesis addresses the problem of dynamically modeling fracture of brittle objects in next-generation games. The challenges of this research are to implement a fracture framework in real-time and that produces results that are visually realistic. An image-based procedural approach is proposed since speed and memory usage are important factors in a commercial game system. The criteria for fracture depend on the strength of the impact force and position of the impact point. The crack patterns are based on a probability texture that defines the weak areas of the object where the crack propagates. The probability textures are different for various materials and are specifically designed by the user. Our technique is unique and allows the user to have more control over the shape of the fragments for different types of objects. The results presented demonstrate that the fragments can be dynamically generated in real-time.

Chapter 1 Introduction

This chapter gives a brief introduction to this research, the scope and objectives. It also gives an overview of the structure of this thesis.

1.1 Motivation

Today, games try to simulate a realistic environment to provide gamers with a fully immersed experience. While most research is aimed at 3D graphics, a new research interest is in realistic physics for games. With increased computational performance of next-generation consoles and computer hardware, more realistic physics simulation in games can be modeled. Physics in games have many uses, including the destruction of objects.

Traditionally, destruction in games is animated in a loop specified by the artists and modelers. As a result, the objects in a game break the same way every time. As the expectation of gamers increases these days, game developers have to use physics to animate a more physically and visually realistic destruction. This involves developing a physics engine with at least rigid body dynamics and collision detection features. However, the objects to be destroyed have to be modeled as a broken object then joined together by links. When the objects breaks, the links are broken and physics laws animate each broken pieces. This method will always show the same broken pieces.

Although there is a huge interest in simulating realistic destruction of models in the fields of material science and animation, only a few are targeted at games.

1.2 Objectives and Scope

The research described in this thesis addresses the problem of dynamically modeling the fracture of brittle objects in next-generation games. The dynamic modelling of a fracture refers to creating fractures on an object that looks different every time it is hit. Figure 1.1 below shows a square concrete wall being fractured differently when hit at different areas. This is to simulate a real-world fracture. The fracture propagation and generation of fragments have to be done in real-time and look visually realistic.

Figure 1.1 Examples of a concrete wall being dynamically fractured.

The aim of the research is to develop a fracture framework that works with the game engine to fracture objects in the game. The model should dynamically fracture objects within the classification of brittle rigid bodies such as concrete, wood and glass. These objects are represented as three-dimensional surface meshes.

1.3 Assumptions

In order to achieve results in real-time, necessary assumptions have to be made. Firstly, we only consider solid brittle objects. Our classifications of brittle objects are as follows:

- Concrete walls / slabs
- Wooden walls / planks

- Glass windows

The objects should have a flat cross-sectional surface that is extruded with a short depth as shown in Figure 1.2 below. The cross-section can either be convex or concave but should not have holes. Currently, we also assume that no deformation takes place when these objects are hit. We consider only the generation of cracks and fracturing of the objects. The cracks generated will also instantly fracture the objects into solid fragments. Once these fragments are generated, the physics engine will handle the dynamics. Our framework will generate these fragments during the course of the game in real-time.

Figure 1.2 Example of objects with flat extruded cross-sectional surface.

1.4 Challenges

Fracture simulation has many uses in different industries. In the engineering world, fracture is simulated with very high accuracy to predict when a structure may fail. The simulation must be very physically realistic to get accurate results so that the engineer can be sure that the structures can withstand stress and strain in the real world.

However, these tests may take a long time to complete due to the complexity. To reduce the time taken, engineering companies may perform this simulation on a cluster of computers or make use of a super computer.

In the world of entertainment, such as in movies and games, the fracture animation need not be physically accurate but instead visually realistic and believable. For movies, the simulation time does not matter since the rendering of the animation also takes time to complete. Therefore, fracture animation in movies can make use of a more physically-based method. For games on the other hand, the fracture animation must be done in real-time. Here, computational time and memory space are limited since the simulation is supposed to be run on consumer hardware.

The main challenge to simulate fracture in games is that it has to be very fast. However, current methods of modeling fractures involve using physically-based methods that result in physically realistic and accurate looking fractures. However, these approaches are computationally expensive for current hardware. Hence, these methods are used mostly for simulation and animation where realism is more important than speed. The methods developed in this research are unique, as they have to be fast and executed in real-time in a game engine. The problem here is that accuracy has to be balanced with speed. To develop a fast method, accuracy has to be compromised. Most of the time, lack of accuracy is acceptable in games.

A technique to fracture objects in the game has to be run during the game itself and has to take a few milliseconds to complete. It is important because in order for the game to achieve a minimum frame rate of 30 frames per second (FPS), the main game loop has to take less than 33.33 milliseconds to complete for each frame. This includes the render time and other computation in the game for a frame such as

artificial intelligence and physics collision. Therefore, the fracture implementation should take just a few milliseconds to complete for the game to run smoothly.

Most games today employ a fracture model that requires the objects to be pre-fractured. As a result, the objects can be fractured in a limited number of ways. Also, the artist will have to spend more time in 3D authoring the internal structure of the objects. Some models require the objects to be discretized into voxels or tetrahedral meshes. Since these objects have an internal structure, they also take up more space in the memory while running the game. Using too much memory space on game objects is not desirable in console games due to limited amount of memory in console hardware.

To sum it up, the challenges of this research are to model fracture that is fast, visually realistic and does not use up too much memory space. These are the same challenges for any feature that is used in next-generation games.

1.5 Methodology

The two main approaches to model a fracture are the physically-based approach and the procedural approach.

The physically-based approach uses numerical methods to simulate the physics of the fracture phenomena [1]. This method produces a physically accurate fracture model but it is usually not fast enough and it requires the objects to be voxelized or tetrahedralized. The physically-based method is commonly used to pre-fracture the objects offline before being loaded into the game.

The procedural approach is proposed in this thesis since speed and memory usage are important factors in developing a game. It is simple and fast, using just heuristics to model the fracture. This approach can create fragments of any shape and size since the objects do not have to be discretized. However, it may not be physically accurate.

The proposed method is an image-based procedural approach that models fracture based on a texture. The texture represents the internal structure of the object by showing the strong and weak areas using probabilities. This gives the artist control over the shape of fragments for different types of objects. A shape mask texture is also used to break objects of irregular shapes.

1.6 Our Contributions

The main aim of this research is to create a fracture framework to be used in next-gen games. The contributions includes

- dynamically fracturing an object during run time,
- small memory footprint for objects that can be fractured,
- flexibility to control crack patterns through probability textures and
- easy creation of probability textures.

Using this fracture framework, the artist can easily and intuitively control the shape of the fragments using specifically designed probability textures. These textures can be easily created using any 2D paint tools.

1.7 Overview

This thesis is organized into six chapters. Chapter 2 provides the background information from relevant fields that is essential to the understanding of this thesis. Chapter 3 discusses most related research to date. Chapter 4 focuses on the main contributions of this thesis, which are the design of a fracture framework for game engines and the procedural technique to model dynamic fractures for brittle objects. Chapter 5 presents the results of the research. Finally, Chapter 6 concludes the thesis with a summary and suggestions.

1.8 Summary

This chapter introduced the idea of creating realistic destruction of objects in games and states the objectives of this research. We also discussed the challenges of modelling fracture in games and the differences between simulating fracture in games and in animation such as movies. We stated that for a technique to be applied in games, it should be fast as well as it does not take up too much memory space. Our proposed method is briefly described and our contributions to the literature with this research were listed.

Chapter 2 Background

The techniques described in this thesis require the understanding of relevant fields from materials science, mechanical engineering and game physics. This chapter gives a brief overview of these relevant fields.

2.1 Fracture Mechanics

In the mechanical engineering field, a fracture is a type of failure of a body by separating into two or more pieces under the action of stress [2]. Fracture mechanics is a study to predict the behaviour of cracked bodies subjected to stress and strain. Methods of analytical solid mechanics, in the field of materials science, are also used to calculate the stress and strain.

2.1.1 Dynamic Fracture Mechanics

The field of fracture mechanics can be divided into a few branches. The two most common ones are elastic-plastic fracture mechanics and dynamic fracture mechanics.

Elastic-plastic fracture mechanics can be applied to materials under quasistatic conditions where the body remains stationary. Here, the cracks formed are stable such as plastic deformation.

Dynamic fracture mechanics [3] is applied to materials under dynamic effect such as inertia. It uses a dynamic system [4] where an unstable force acts on a cracked body

which then acquire kinetic energy. The method presented in this thesis aims to approximate these effects using a probability texture.

2.1.2 Brittle Fracture

The material properties of an object affect the type of fractures created [5]. The two basic types of fracture are ductile fracture and brittle fracture. While ductile fracture occurs after the body goes through a high degree of plastic deformation, brittle fracture does not involve any deformations.

Brittle fracture occurs on materials that fail under tensile stress [6-8]. This type of fracture causes rapid propagation of cracks at high loads and absorbs very little kinetic energy. For example, a brittle object will instantly break apart when hit by an object. This action induces a tensile force on that brittle object. During this fracture, bifurcation also occurs.

The type of brittle fracture addressed in this thesis is also known as rapid fracture where the fracture occurs over a short time span and often leads to the fragmentation of the object [9]. Therefore, we assume that during an impact the object will shatter into pieces instantly without any deformations.

2.1.3 Strength of Materials

A brittle object fractures when the tensile stress subjected to it reaches its tensile strength. The tensile stress, σ is the force per unit area measured in pascals (Pa).

$$\sigma = \frac{F}{A} \quad (1)$$

The two main types of tensile strength are yield strength and ultimate strength [10].

Table 2.1 below shows the tensile strength of some materials [11].

The yield strength is the stress at which the material will start to plastically deform. Before reaching the yield strength, the material will deform elastically. However, this usually occurs in ductile materials. Brittle objects do not have yield strength.

The ultimate tensile strength of a material is the maximum tensile stress it can withstand before it ruptures. For brittle objects discussed in this thesis, their ultimate strength is very low. Hence, these objects break even with a low impact force.

Table 2.1 Strength table of specific materials.

Material	Yield Strength (MPa)	Ultimate Strength (MPa)
Glass	-	50 (in compression)
Marble	-	15
Concrete	-	3
Pine Wood	-	40
Structural Steel	250	400
Titanium Alloy	830	900
Stainless Steel	520	860

2.2 Crack Generation and Propagation

The aim of this research is to create visually realistic fragments. Therefore, we have to study how brittle objects break in real life. Cracks propagate differently for different types of brittle materials, creating different types of crack patterns [12]. For example, cracks on a brick wall looks very different from a concrete wall. As discussed by

Abraham [13], the features of a fracture simulation depend on the crystal orientation of the object. The cracks should propagate along the weakness areas of the material.

Figure 2.1 below shows real cracks on brick walls. The cracks follow the structure of the wall and propagate through the mortar areas avoiding the bricks and have a zigzag pattern. Sometimes, if the tensile stress is large enough, the cracks may even propagate through a brick breaking it into half. A brick wall can usually resist compressive stress the best but not tensile stress.

Figure 2.1 Cracks on brick walls.

Figure 2.2 Cracks on concrete.

The images in Figure 2.2 above show cracks on concrete walls. The crack pattern here is more arbitrary and the cracks may bifurcate. This is because concrete has a mostly homogenous structure if it is not reinforced. Unlike the crack pattern on a brick wall, the cracks on concrete can propagate to any directions and can be in any patterns.

Figure 2.3 below shows that the cracks in wood follow the grain of the wood. This pattern is usually parallel lines or curve lines that start from a point. The images at the top show the cracks on a wooden fence. It appears straight since the grains of that plank are straight. The start of the crack is from the nails that hold the wooden planks together. The two bottom images of Figure 2.3 show the crack pattern on the cross section of a tree trunk. Most of the crack patterns are radial and look like a magnetic field.

Figure 2.3 Cracks in wood.

Figure 2.4 Cracks on glass.

Figure 2.4 above show one type of crack pattern on glass. This is a static crack pattern that can be found usually in glass and ceramics. Note that the cracks generally meet at an angle close to 90 degrees. Also note that the crack does not cross each other because tension cannot cross a crack. The crack will usually bifurcates and reach the edge of the object at about a 90 degrees angle.

2.3 Physics in Games

Physics have started to play an important role in games to add depth and make it feel more realistic. Although many games use physics engines, most of them does not have the full features of a physics engine [14]. The models for use in the game have to be assigned physical attributes to perform physical simulation on it [15]. The most few important features of a physics engine are kinematics and dynamics.

Kinematics deals with the movement of objects over time. It is determined by the position, velocity and acceleration of the object without considering the forces acting on it [16]. There are two type of kinematics, inverse and forward. Forward kinematics applies specific conditions to compute the end position of the objects. Inverse kinematics however, is the opposite where the end position is specified to compute the conditions to achieve the goal. Inverse kinematics is mostly used in character animation and is computationally expensive. Forward kinematics is widely used for animation in games.

Dynamics is the study of forces and masses that cause objects to move over time. The kinematics quantities will be computed based on initial conditions. The few types of dynamics are fluid dynamics, soft body dynamics and rigid body dynamics. Fluid dynamics refers to the dynamics of liquids or gases such as water. Soft body dynamics refers to the dynamics of objects that deforms during the simulation while rigid body dynamics is performed on objects that do not deform. This thesis considers only brittle solids so only rigid body dynamics is considered.

Currently, it is computationally expensive to apply all the real-world maths and physics in games. As a result, the mathematical computation in a physics engine for games is

simplified to reduce computational load as long as it looks realistic enough and runs fast.

2.3.1 Rigid Body Dynamics

Rigid body dynamics refers to the dynamics of objects that have masses and have a fixed shape throughout the simulation. Articulated figures such as a human body, rigid bodies connected by joints, can be governed by rigid body dynamics. [14, 17-19] are good references on rigid body dynamics in games.

The state vector of a rigid body is defined as

$$Y(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}. \quad (2)$$

This state vector is the position, orientation, linear and angular momentum of a rigid body [20]. The position vector is $x(t)$. The orientation $R(t)$, is a 3 x 3 rotation matrix.

The functions are with respect to time, t .

$$R(t) = \begin{pmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yz} & r_{zz} \end{pmatrix} \quad (3)$$

The linear momentum $P(t)$, is the multiplication of the mass M and the linear velocity $v(t)$.

$$P(t) = Mv(t) \quad (4)$$

The angular momentum $L(t)$, is the multiplication of the inertia tensor $I(t)$ and the angular velocity $\omega(t)$. The inertia tensor $I(t)$, is a 3×3 symmetric matrix that is based on the mass of the rigid body. I_{body} is the inertia tensor in body-space of the object.

$$I(t) = R(t)I_{body}R(t)^T \quad (5)$$

$$\omega(t) = I(t)^{-1}L(t) \quad (6)$$

$$L(t) = I(t)\omega(t) \quad (7)$$

The derivative of the state vector is

$$\frac{d}{dt}Y(t) = \frac{d}{dt} \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t)q(t) \\ F(t) \\ \tau(t) \end{pmatrix}. \quad (8)$$

This derivative is computed by Euler integration to solve the state vector of the rigid body. This state vector is computed for every time frame to render the object at the correct position with the correct orientation.

2.3.2 Collision Detection and Collision Response

Collision plays an important role in a physics engine. Without it, the objects may interpenetrate each other and will not look realistic. The two phases of collision are collision detection and collision response [21]. While collision detection checks for

intersection between two objects, collision response decides the reaction of these two objects if there is a collision.

Intersection tests on the geometry of two objects itself are very expensive, especially if the objects are concave and have many triangles. To increase performance, overlapping tests on the objects' bounding volumes are done before the actual intersection tests [22]. The main types of bounding volumes are a bounding box, sphere and convex hull. These bounding volumes are primitive convex shapes that allows simple overlapping test on them. Bounding volumes that are tighter fitting and have better culling performs faster overlapping tests but are often harder to implement. They also take up more memory and computational cost. Figure 2.5 shows the different types of bounding volumes.

Figure 2.5 Types of bounding volumes.

The bounding sphere is the easiest to implement and has the fastest performance. To do overlapping tests on two spheres only 4 scalar values are needed; the center of each spheres and their radii. To check if the spheres overlaps, the Euclidean distance between the center of both spheres is computed and compared to the sum of the sphere radii [22].

The axis-aligned bounding box (AABB) is one of the most common bounding volumes [22]. An AABB is a rectangular six-sided box where its edges are always parallel to the axes of the coordinate system. It features a fast overlapping test and is easy to compute. Each AABB is represented with six scalar values. The overlapping test for AABB just involves comparing the six values of each AABB. The only disadvantage of an AABB is that it has to be recomputed when the object rotates.

The oriented bounding box (OBB) is also a rectangular box like an AABB but has an arbitrary orientation. Representing an OBB is quite expensive since usually 15 floats is used for each OBB [22]. However an OBB has a tighter fit compared to an AABB and its orientation follows the object itself. The test for an overlap of two OBB is done with a separating axis test that is quite complicated.

A convex hull is a convex polyhedron of the object. It has a very tight fit and follows the orientation of the object. Therefore, it does not have to be recomputed when the object rotates. However, the convex hull itself and intersection tests between convex hulls are expensive to compute. Convex hulls also require more memory to store its surfaces. Algorithms used to compute a convex hull includes Gift Wrapping, Divide and Conquer and the Incremental Algorithm [23].

Once collision is detected, the physics engine will decide what happens next. The collision detection algorithm often at least returns the point of impact and the forces of impact between two objects [24]. Usually, for rigid bodies, the response will be the transfer of resultant forces to each object making them move away.

2.4 Summary

In this chapter, we discussed the use of fracture mechanics for simulation of fracture. A basic knowledge of fracture dynamics is required to understand this research. Since this research is aimed towards the use in games, we also discussed on how physics is simulated in games. Though our research does not include the implementation of rigid body dynamics and collision detection, it is necessary to understand it as our framework layer lies on top of a physics engine.

Chapter 3 Related Work

This chapter discusses the current literature survey of work related to fracture modeling. It is split into the two main approaches of physically-based modeling and procedural modeling.

3.1 Physically-based Modeling

Most work in this field has been focused on physically-based method to model fracture. The following sections discuss the different sub-models that use the physically-based method. This approach usually requires the body to be discretized to voxels or elements forming a tetrahedral mesh.

3.1.1 Finite Element Model

The most common system to model fractures is using the finite element method [25, 26]. Even though the finite element method is mostly used in the engineering field to achieve very physically realistic results, it can also be used in animation to get similar results.

Before using this method, the object has to be tetrahedralized from triangle meshes shown in Figure 3.1 below. Cutler et al. [27] presented a procedural approach to convert solid models to tetrahedral representations. They synthesize tetrahedral models from triangle meshes by evaluating the signed distance field on a uniform 3D grid.

Figure 3.1 A tetrahedralized mesh.

The finite element method is a powerful system to model fractures and deformations accurately but it is not fast [28-32]. Once the material has been discretized into disjoint elements, the material derivatives are computed from shape functions that interpolate the material within the element.

Zhaosheng et al. [28] fractured brittle objects that are rigid bodies endowed with a triangulated surface mesh, a level set volume for collisions and a tetrahedral mesh for finite element analysis. They use an impulse-based solver for the rigid body simulation. The results of the simulation look very realistic as they also simulate ductile fracture but it is not fast.

Müller et al. [33] presented a method for physically-based animation of surface meshes by generating cube meshes as the elements of a finite element model. The results show a cube mesh of about 800 elements being generated at 10 to 15 frames per second. The cube mesh of an eagle is shown in Figure 3.2. A simulation to fracture a pig model was stated to run at more than 40 frames per second but the complexity of the mesh was not stated.

Figure 3.2 An eagle represented as a cube mesh.

O'Brien and Hodgins [29, 30] used the finite element model and analyse the stress tensors created to determine the direction of the crack propagation. The tetrahedral mesh is dynamically retessellated at every arbitrary propagation step so that the cracks generated are not limited to the element boundaries. Therefore, irregularly shaped shards and edges will be formed when they shatter. Their results are shown in Figure 3.3.

A similar approach was taken by Müller et al. [34], although the finite element formulation was simplified to achieve performance in real-time. They presented a stable hybrid method for simulating deformation and fracture of stiff materials. However, this approach is still not fast enough to be employed into games.

Figure 3.3 Destruction of a wall with a finite element model.

3.1.2 Spring-mass Model

Another approach uses the spring-mass model that can model both fractures and deformation. In 1988, Terzopoulos and Fleischer introduced a general technique for animating viscoelastic and plastic deformations [35]. They model the fracture by breaking the links between two nodes when the distance between them exceeded the threshold. Their results showed the tearing of cloth and sheets of paper. Norton et al. [36] also used a network of point masses connected by springs to represent objects that can deform or fracture. They presented their results by breaking a teapot.

This spring network model performs faster than the finite element method above and works best for deformations. Hirota et al. [37] used this model to simulate cracks on a drying clay. Steinemann [38] managed to achieve 30 to 40 frames per second using a spring-mass model of 3000 vertices.

3.1.3 Point-mass Model

A fast approach to simulate brittle fracture uses the point-mass model. Smith et al. [39] presented a method for the rapid and controllable simulation of the shattering of brittle objects that are represented as point masses connected by distance-preserving linear constraints. However, this model does not compute deformations. The result of this model is shown in Figure 3.4 below. The wine glass was modelled with 3422 tetrahedra and took about 2 seconds per impact step and about 30 seconds to generate the fragments.

This model has realistic looking results but the mesh still needs to be tetrahedralized. A high-resolution tetrahedral mesh is needed to have less “jaggy” effects on the fragments but this increases the computational costs.

Figure 3.4 Results of a shattered glass with a point-mass model.

3.1.4 Explosion Model

Some techniques have been developed to simulate exploding objects. Firstly, a voxel-based approach represents objects in voxels connected by links between adjacent voxels that breaks when the pressure at each link exceeds the yield limit [40-43]. Mazarak et al. [40] presented an explosion model that results in realistic solid debris with a spherical blast wave. Martins et al. [41] extended Mazarak's work to achieve real-time performance. He also improved the voxel model by having arbitrarily shaped voxels to make object modeling more flexible. Auxiliary visual effects were added to increase the visual believability of the explosions without making it more computationally expensive. Yngve et al. [42] presented a physically-based approach to model explosions with visual effects such as fireballs and dust clouds. This approach used the Friedlander equation to approximate blast waves.

Ek et al. [44] presented a framework to animate real-time explosions completely on the GPU. A three dimensional computational fluid dynamics (CFD) was used to achieve results of about 30 frames per second with a grid of size 32 by 64 by 32. The implementation was done on GPU with DirectX 10 using HLSL fragment shaders.

Next, a rapid fracture approach works at a short time scale at which explosion occur. The cracks propagate quickly under high loads causing it to fracture. Neff and Fiume [9] presented a blast wave simulation using a set of blast curves which are a simplified set of equations to compute the distance of the object from the center of explosion. They also have a rapid fracture model to generate cracks that propagate at high speeds and bifurcate. However, their algorithm is limited to thin surfaces and is not easily extensible to three-dimensional solids.

3.1.5 Static Crack Patterns

There is also research on generating static crack patterns that use the physically-based model. Gobron and Chiba [45, 46] used a stress spectrum model on a 3D surface cellular automata. Their method is semi-physical and allows any type of cracks to propagate on any type of triangulated 3D surface.

Iben and O'Brien [47] used a first-order quasistatic system to generate surface crack patterns on mud, ceramic glaze and glass from a stress field defined heuristically over a triangle discretization of the surface. The user has control over the crack patterns by changing a set of simple parameters. The result looks very close to real crack patterns because it is physically-based but it is not fast.

Hirota et al. [48] used a spring network system to model surface cracks. The cracks are generated by cutting the springs connecting the elements. This model works for 3D surface only.

Federl and Prusinkiewicz [49] used the finite element method to simulate fracture formation on growing surfaces. The mesh is refined along the lines of fracture and it takes a few hours and between 60 to 150 elements to generate the crack patterns.

3.2 Meshless Model

A trend to enhance the finite element method is to use meshless methods [50-52]. This method avoids remeshing the models and solves problems with element cutting during fracture simulation. This model can also be used for simulating deformation.

Pauly et al. [51] used a point-based representation to model the initial object and broken objects. The boundary of the object is defined by a set of surface sheets called *surfels*. A brittle fracture of an object was simulated at 22 seconds per frame with 249 thousand initial surface samples. 61 thousand surface samples were created during the fracture forming a higher level of surface detail of the broken pieces.

3.3 Procedural Modeling

Procedural modelling is an approach to models fracture without using any physically-based methods such as finite element method. This approach may be less physically accurate but may be fast.

Martinet et al. [53] presented a method that lets the designer have control over the pattern of the cracks and the size and shape of the fragments generated. First, the designer created a 2D crack pattern using an editor. This crack pattern is the shape of the cutting tool that will cut the object at every step. This crack pattern can be a simple jagged line to showing the edges of the “saw”-like cutting tool. Next, this crack pattern is mapped on the surface of the object by a surface marching algorithm to generate a

geometric skeleton. Then the carving volume is generated by sweeping the profile curve along the skeleton. The fragments are then created by using Boolean operations between the original object and the carving volume [54]. Here, the original object is converted into a point cloud representation as a pre-processing step. A Hybrid Tree model [55, 56] was used to store the skeletal implicit surfaces and the triangle meshes. This tree may be polygonized for fast visualization but the algorithms cannot be run in real-time. The results from this method are shown in Figure 3.5 below. While our method creates a fracture pattern once, Martinet's method cuts the object at every step using the cutting tool.

Figure 3.5 Results from procedurally modeling of fracture.

Mould [57] synthesized controllable crack patterns by using images. His algorithm is based on the Voronoi diagram of a weighted graph. Similar to our approach, his crack patterns are generated in image space. However, his work is aimed at the animation and non-photorealistic rendering of crack texture. The objects that are to be fractured do not shatter into pieces. He achieved a time of 0.9 seconds for a 400 by 400 image.

Wyvill [58] presented an algorithm to simulate cracks found in Batik wax painting by using non-photo realistic rendering. It is based on the Distance Transform algorithm [59] that is not physically-based. It takes a few seconds to generate the Batik patterns.

Raghavachary [60] generated brittle fracture on polygonal surfaces by synthesis of Voronoi polygons on each face of a polygonal mesh. The cracks are propagated along the edges of the Voronoi polygon networks.

Eberle [61] presented a non-physical based approach to create a subset of impact damage effects on brittle objects. The initial object is a tetrahedral mesh and the elements within a radius of the impact point will be active. The damage value is computed based on the distance to the impact point.

A thesis by Workman [62] presented a procedural system with two parts. The offline part of the system generates a 2D then 3D crack pattern based on an alpha texture. The real-time part loads the 3D crack pattern and applies it to a target object. Although this system was aimed for usage in games, it was not tested in a game environment and the offline part of the system was not fully completed. Workman's work was based on two other Master thesis by Keith [63] and Fox [64]. However, his work shows a unique approach of generating crack patterns using an alpha texture.

Oda and Chenney [65] presented a multi-stage refinement method that reduces the computational cost compare to Smith et al. [39]. They subdivided the triangles near the collision point, giving more details to the fractured areas instead of discretizing the whole object. They achieved results of 60 FPS generating 150 triangles with 240 constraints. Oda then extended his work to from 2D extruded surfaces to 3D surfaces [66]. The fracture algorithm with 3800 constraints took 2.3 seconds to complete.

3.4 Summary

Table 3.1 below shows the comparison of current methods to model fracture. However, most methods are not able to run in real-time as they are not targeted towards games application.

Table 3.1 Comparison of current methods.

Author	Method	Object Representation	Time to generate fracture
Steinemann (2005)	Mass-spring model	Thick surface mesh	40 FPS for 3000 vertices (Cow)
Müller et al. (2004)	FEM	Cube mesh	40 FPS (Pig)
Ek et al. (2007)	GPU	Particles	30 FPS with 32x64x32 grid texture (23000 particles)
Oda and Chenney (2006)	Multi-stage refinement scheme	Tetrahedralized mesh	2.3 seconds with 3800 constraints
Martinet et al. (2004)	Hybrid Tree model	Point cloud	5.96 seconds to create 48 fragments (Flute glass)
Pauly et al. (2005)	Meshless model	Point-based surface	22 seconds per frame (249k surface samples)
Smith et al. (2001)	Point mass model	Tetrahedralized mesh	Up to 30 seconds to generate fragments with 9000 constraints.

The top three authors in the table above showed methods that can be run in real-time, while others are fast but could not achieve a real-time performance. Steinemann [38] and Müller et al. [33] achieved real-time performance however, these objects have to be discretized. The limitation for discretized objects is that they take time to be generated. Also, these objects have a large memory footprint when loading them in games. As described in Chapter 1, one of the challenges for game development is the memory limit. The game will take a longer time to load if many objects contain many

triangles. Oda and Chenney [66] and Smith et al. [39] work also have the same limitations to having a large memory footprint for tetrahedralized objects.

Ek et al. [44] presented a novel approach of using GPU to simulate explosions and achieved real-time performance. However, their work is to simulate explosions by animating particles. Their technique of using computational fluid dynamics did not show any simulation of fracture.

Both Martinet et al. [53] and Pauly et al. [51] used the point-based representation of objects. This method is slow for games since these objects will still have to be converted to triangles to be rendered in game engines.

As described in this section, there are some work that can simulate fracture in real-time but there are limitations for their usage in games such as the large memory footprint. The work in this survey does not show any results in an existing game or game engine. Although there are current implementations in games showing real-time fracture, such as the Havok engine and Crytek engine, little is known about their methods to simulate fracture. Our research aims to contribute to this gap to show how fracture simulation is used in game applications.

Our method of simulating fracture is different from current real-time approaches. Our approach is based on the Mould's [57] idea of using an image to guide the fracture. The difference of our approach compared to Mould's is that our image shows the strength of the material while Mould's image is just any drawing. We are also able to achieve real-time performance whereas Moulds's technique took at least 0.9 seconds and did not completely fracture the object.

Chapter 4 Fracture Framework

This research aims to model fracture of brittle objects in games dynamically and in real-time. To achieve this, we need a framework that can be fitted into a game engine. This chapter presents the fracture framework and the approach it uses.

4.1 Fracture Framework Overview

The fracture model presented in this thesis is based on a procedural modeling approach. We simulate fracture of different types of materials by propagating cracks on the surface of the objects based on its probability texture. This method is not physically accurate but simulates real-world fracture by generating 2D crack patterns. This method can also create various kinds of fracture that are image based.

The framework starts when a collision is detected on a breakable object. It is then divided into three stages as shown in Figure 4.1. In the first stage, the impact point and impact force are used to decide if the fracture should be initiated. This is further explained in Chapter 4.3. The next stage will run the 2D crack propagation algorithm and create a crack pattern in a bitmap and a graph. The crack pattern will be generated using a probability texture that affects the crack propagation and a mask texture that defines the boundary shape of the original object. The crack propagation process will be discussed in Chapter 4.4. In the third stage, the crack pattern will be extruded into a 3D mesh. The 3D mesh generation is discussed in Chapter 4.5. The framework then generates the 3D models of the broken fragments of the original object. These fragments will then replace the original model and the physics engine will animate these broken pieces. This framework will only work for objects that are

defined as breakable and are in the classification of breakable brittle objects in Chapter 1.3.

Figure 4.1 Flow diagram of fracture framework.

4.2 Breakable Material

4.2.1 Material Settings

Each breakable object has to be assigned a material. This material will affect how the object breaks. The properties of each material include the strength threshold value, the probability texture and the mask texture. Below is an example of the material settings file. Each parameter will be explained in the next sub-sections.

```
[Concrete]
strength_threshold = 140.0
probability_tex = "concrete_064.bmp"
shapemask_tex = "concrete_mask_064.bmp"

[Wood]
strength_threshold = 100.0
probability_tex = "wood_064.bmp"
shapemask_tex = "wood_mask_064.bmp"

[Glass]
strength_threshold = 80.0
probability_tex = "glass_064.bmp"
shapemask_tex = "glass_mask_064.bmp"
```

Figure 4.2 Material settings file.

4.2.2 Strength Threshold

The first parameter is the strength threshold of the material. This value is based on the fracture toughness of the material [11]. The strength threshold is also based on the game engine and should be tweaked according to the fracture toughness of the material. An example of the strength threshold values of different types of materials is shown below. These values are used to decide if the object will be fractured.

In our example, there are no units for the strength threshold value and it is roughly calculated as 100 times the fracture toughness of its material. The reason for this is because the impact force value retrieved from our physics engine varies from about 10 to 1000. If the strength threshold value is as low as the fracture toughness value, objects of all materials will probably be fractured. By adding a factor of 100 or more, different levels impact force will cause different materials to fracture. For our implementation, the strength threshold value should have a clear variation for different materials to show which materials will fracture under little force while other material should fracture under great impact force. This identifies the strength of the material.

Table 4.1 Strength threshold of material.

Material	Fracture Toughness ($\text{MPa}\sqrt{m}$)	Strength Threshold
Glass	0.7-0.8	80
Concrete	0.2-1.4	140
Wood	0.5-1.0	100
Acrylic	1.0	100
Polystyrene	0.8-1.1	110
Steel Alloy	50	5000

As shown in Table 4.1 above, steel objects have the highest threshold value and would not break easily compared to glass, concrete or wood. The strength threshold value is the limit of the force value acting on the object that the material can withstand without breaking. This value will be used in the fracture initiation process.

4.2.3 Probability Texture

The probability texture is the main data that affects how the object breaks. It is used to decide the direction of the cracks hence creating a crack pattern. The specifications of the probability texture are shown below:

- 2D grayscale bitmap.
- 8 bit per pixel.
- Size of between 32x32 pixels to 128x128 pixels.

We use the 8-bit pixel value as a probability value for each pixel. Bright areas have a larger pixel value with a maximum of 255 while the dark areas have a value nearer to 0. Higher probability pixels will allow the crack to propagate more to that area. The bright areas on the probability texture indicate this. The bright areas also indicate the weak areas of the material. It is assumed that crack will propagate through the weak areas of the material. This also simulates the crack in real life since every material has an internal structure with some weak points. For example, the grain lines of wood are the weak areas of the wooden material since wooden objects usually break along its grains. To simulate this, we can create a probability texture similar to the grain lines of wood shown in Figure 4.3 below. The probability textures shown below ranges from 32x32 pixels to 128x128 pixels.

Figure 4.3 Photo of grain lines on wood and probability textures of wood.

We simulate real cracks of wood and glass by creating probability textures based on photos of cracks on the material. Examples of probability textures for glass and concrete are shown below.

Figure 4.4 Photo of cracks on glass and probability textures of glass.

Figure 4.5 Photo of cracks on concrete and probability textures of concrete.

Any paint tool such as Adobe Photoshop can create these probability textures. First create a black image then draw white crack lines with a 1 pixel sized brush. To create blended gray lines, the hardness of the brush can be reduced. A simplest probability texture could just be a noise image as shown in Figure 4.6 below. This probability texture will create cracks at random since there are no main weak areas for the crack to propagate.

Figure 4.6 A probability texture with noise.

Other than probability textures created based on real cracks, we also have the flexibility to create cracks based on any arbitrary image. A crack can be designed by the user. We can crack an object in the shape of letters, numbers or any imaginative shapes. Some examples of custom cracks are shown in Figure 4.7 below.

Figure 4.7 Custom probability textures.

4.2.4 Shape Mask

The shape mask texture is used to break an object that is not a cuboid. This texture is an alpha mask of the cross-sectional surface of the object. The specifications of the shape mask are shown below:

- 2D grayscale bitmap.
- 8 bit per pixel.
- Size of between 32x32 pixels to 128x128 pixels.

Although it is a grayscale bitmap, the background colour should be a 100% black with the mask colour should be 100% white. However, the edges of the mask can be blurred to have a better smoothing effect. Examples of different shape masks are shown in Figure 4.8 below.

Figure 4.8 Shape masks of different objects such a heart-shape and bunny.

The shape masks can be created with any paint tool. However, the easiest way is to create it straight from a 3D modeling tool that the object is modeled from. For example, if the object is modeled in 3D Studio Max, a simple render from the front view will create this mask where the background is black and the object is white. The screenshot of the model and its shape mask is shown in Figure 4.9 below.

Figure 4.9 Creating shape masks from 3DS Max.

In our current implementation, the shape masks can be of any arbitrary shape with no holes. Shapes with genus or holes are not supported.

4.3 Fracture Initiation

Our pipeline starts with the fracture initiation stage to check whether the object will be broken. This happens when a breakable object is hit. The impact point as well as the impact force is retrieved from a collision detection algorithm. These parameters are generated from the existing physics engine in the game. A probability texture is then mapped onto the surface of impact of the object shown in Figure 4.10 below. The square shaped probability texture is stretched to the four edges of the object with a rectangular cross-section. If the object is arbitrarily shaped, the probability texture is mapped to the surface of the bounding box that is hit.

Figure 4.10 Mapping of probability texture to object.

The pixel value, p , is then read that has the coordinates that corresponds to the impact point. The strength threshold value, t , of the object's material is retrieved. Using the formula below, the fracture will be initiated only if the value of the impact force, f , is greater than the calculated threshold value.

$$f > t \times \left(1 + \frac{128 - p}{255} \right) \quad (9)$$

The pixel value on the probability texture is converted to a value between 0.5 and 1.5. This value is then multiplied to the strength threshold value to increase or decrease it depending on the strength of the material. If the impact point is on a pixel with value 128, the overall threshold value will be equivalent to the strength threshold value of the material. The probability texture is used to increase the probability of fracture of stronger objects if its weak area is hit.

4.4 2D Crack Propagation

4.4.1 2D Crack Pattern Test Program

This section presents the crack propagation algorithm. This is the core of the fracture framework that generates a crack pattern from a probability texture and its material settings. A simple program was implemented to test the 2D crack propagation algorithm before putting it in the fracture framework. It is also useful to have a visual representation of the crack pattern in 2D as shown in Figure 4.11 below.

Figure 4.11 Screenshot of crack propagation test program.

In this program, the top left box shows the probability texture while the top right box shows the shape mask. The bottom two boxes show the output of the algorithm. The bottom left is the bitmap grid of the crack pattern while the bottom right is the crack pattern stored in a graph. The impact point of start of the crack is indicated by the hand cursor on the top left box. The next sub-sections will explain the 2D crack propagation and the uses of the program.

4.4.2 Data structure

The 2D crack pattern that is generated from this algorithm is stored in a graph data structure shown in Figure 4.12 below. It is a closed directed graph with the impact point as the root. Every node on the graph represents a vertex on a broken piece. The nodes within the mask area are the crack lines that are generated. A graph data structure is chosen so that the connectivity between the crack branches is preserved and a simple graph travel is needed to create the broken fragments.

Figure 4.12 Graph representation of the crack pattern.

Another bitmap grid is also used to mark areas where the cracks have been propagated. As shown in Figure 4.13 below, the propagated crack branches are marked as red while the boundary outlines is marked as black pixels. This data is essential to prevent cracks from crossing each other. In our implementation, we store data in the bitmap to help decide in the direction of the crack propagation and the termination of the crack branch. This will be discussed in the next few sub-sections.

Figure 4.13 Bitmap representation of the crack pattern.

4.4.3 Getting boundary mask of object

The first procedure is to get the boundary nodes of the object from its shape mask texture. The boundary nodes are stored in the graph at first and define the edges where the cracks should stop propagating. This process has two steps; the first is to find the first boundary node and the second is to propagate a line through the edges of the object's shape mask.

To find the first boundary node, every pixel of the shape mask is checked from the top left, row by row. A simple edge detection check is use on each pixel until a minimum value of 255 is found. This value shows that the neighbouring pixels have the most contrasting value. From Figure 4.14 below, *X* is the pixel being checked and 0, 1, 2 and 3 are the four neighbouring pixels. If a neighbour pixel that is out of bounds, it is assigned a value of 0, which is black.

Figure 4.14 Neighbouring pixel for simple edge detection.

The criterion for the first boundary node is shown below, where p_0 , p_1 , p_2 and p_3 are the pixel values of the neighbour pixel 0, 1, 2 and 3 respectively.

$$\sqrt{(p_0 - p_1)^2 + (p_2 - p_3)^2} > 255 \quad (10)$$

The next step is to traverse the edges of the boundary mask till it reaches the first boundary node. This procedure checks the eight neighbours of the starting pixel to find their edge detection values similar to Equation 10 above. The eight neighbours of pixel *X* are numbered as shown in Figure 4.15 below. Each pixel that is traversed will be stored in the graph creating a loop of the boundary mask. Each pixel is also marked as a black coloured pixel on the bitmap grid.

3	4	5
2	X	6
1	0	7

Figure 4.15 Neighbouring 8 pixels for boundary mask traversal.

4.4.4 Crack propagation

The crack propagation algorithm uses simple heuristics to propagate the cracks hence forming a crack pattern. It starts from the impact point of the collision then propagate the cracks to the edges of the object. The basic idea is to propagate the crack along the weak areas of the object represented by the brighter areas on the probability texture. The main steps of the crack propagation algorithm are shown below. These steps run in a loop until all crack branches are terminated.

1. Calculate probability of neighbouring pixels.
2. Select one or two neighbours with the highest probability to propagate the crack to. If two pixels are selected, new branches are formed.
3. Repeat until each crack branch reaches the edge or intersect with another existing crack.

Step 1: Probability calculation

For step 1, 24 neighbouring pixels shown in Figure 4.16 below will be checked and their pixel probabilities are computed. Neighbours 0 to 7 are the first degree neighbours while the 16 neighbours numbered 8 to 23 are the second degree neighbours. The second degree neighbours are checked to decide the direction of

propagation while both degree neighbours are checked for termination shown in Figure 4.17 and Figure 4.18 below. The second degree neighbours is used instead of just the first degree because of having more freedom for the crack to propagate. This will be explained in Chapter 5.1.1 later.

14	15	16	17	18
13	3	4	5	19
12	2	X	6	20
11	1	0	7	21
10	9	8	23	21

Figure 4.16 Neighbouring pixels for crack propagation.

The pseudo codes for pixel probability computation are shown below.

```

If (pixel is out of probability texture) &&
  (pixel is on another branch)
{
    Set pixel probability to -1
}
Else
{
    Set pixel probability to 1
}

```

Figure 4.17 Pixel probability pseudo code for first degree neighbour.


```

If (pixel is out of probability texture) &&
  (pixel is on another branch)
{
    Set pixel probability to -1
}
Else
{
    A = Get neighbour pixel value from probability texture
    B = Get value based on cosine of angle between the
    neighbour pixel and the previous pixel
    C = Get value based on cosine of angle between the
    neighbour pixel and the start of the branch
    Set pixel probability to (A * B * C)
}

```

Figure 4.18 Pixel probability pseudo code for second degree neighbour.

In Figure 4.18 above, the total pixel probability of the second degree neighbour is computed in three levels. The first level is to get the pixel value from the probability texture. This value ranges from 0 to 255. The second and third level requires the computation of the cosine of the angle between two vectors. These two levels are necessary to prevent backward propagation of the cracks and to lead the cracks to propagate to the edges.

In the second level of probability calculation, the cosine of the angle between the neighbour pixel and previous pixel is computed using the dot product of the vector to the next neighbour and to the previous neighbour from the current pixel as shown in Figure 4.19 below. The neighbour pixel refers to the second degree neighbour of the current pixel that is currently being computed. At first, each of the neighbours has the same probability to be propagated to. Assumed that the crack has propagated a few times and it is propagating to the right.

The solid line represents the current direction of the crack. The dashed line is the direction to the neighbouring pixel is being computed.

Figure 4.19 Angle between previous and next neighbour.

The cosine of an angle between two vectors can be calculated by taking its dot product divided by its magnitude each, defined by the formula below, where a and b are two vectors and θ is the angle between them.

$$\cos(\theta) = \left(\frac{a \cdot b}{|a||b|} \right) \quad (11)$$

This will give a value of 0 when the angle is 90 or 270 degrees shown in Figure 4.20 below.

Figure 4.20 A normal cosine function.

However, we want a low value when the angle is around 0 degrees and high when the angle is about 180 degrees. The equation to the function is shown below.

$$y = -\cos(x) + 0.5 \quad (12)$$

This function gives an output of -0.5 to 1.5. Figure 4.21 below shows the graph of Equation 12. The neighbour that is directly opposite to the previous propagated pixel will have the highest probability value. The neighbouring pixel that is further away from the previous neighbour will have a higher probability than the neighbouring pixel nearer to the previous neighbour. This will prevent the crack from propagating backwards or loop backwards. The purpose of this check is also to propagate the crack away from the start point. A negative probability indicates that the crack will never propagate to that direction of the neighbour pixel.

Figure 4.21 Cosine function of Equation 12.

In the third level of probability calculation, the cosine of the angle between the next neighbour and the start of the branch is computed as shown in Figure 4.22 below. The start of the branch refers to the current branch that the crack propagates from. The solid lines show the previous few directions that the crack has propagated to while the dotted line is the direction to the start of the crack branch. The dashed line points to the next neighbour pixel that is being computed. This calculation will cause the crack to propagate away from the start of the branch.

Figure 4.22 Angle between start of branch and next neighbour.

The equation to get the probability of the angle between the next neighbour and the start of the branch is shown below.

$$y = -0.5 \cos(x) + 0.5 \quad (13)$$

The output of the function above is between 0 and 1. This function gives a smooth gradient of probabilities to the neighbours with the neighbours that are in a direction away from the start of the branch a higher probability. Figure 4.23 below shows the graph of Equation 13. All neighbouring pixels will have a probability to propagate using this function but the neighbours that are in the direction towards the start of the branch will have very low probability. The difference between equation 12 and 13 is that equation 12 will assign negative values to pixels that the crack should not propagate to while equation 13 will still allow the cracks to propagate towards the start of the branch but with a very low probability.

Figure 4.23 Cosine function of Equation 13.

To calculate the total pixel probability of the neighbouring pixel, the probability values taken from the probability texture is multiplied by the probability values from the two

functions described above. The total probability values of each second degree neighbour are then stored in a list and sorted by descending order.

Step 2: Selection of propagation direction and bifurcation

While the first step of the crack propagation algorithm is to assign the probabilities for each second degree neighbours, the second step is to select the neighbours to propagate the crack to. The neighbour pixels in this step are defined as the second degree neighbours. Every crack will propagate to at least one neighbour and bifurcate to at most two neighbours. An exception is for the start of the crack. At the start, the crack will bifurcate into two to three crack branches. This is because we need at least 2 crack branches to complete a fracture with at least two fragments. The procedure to propagate the first branches from the start point is shown below.

1. Propagate to neighbour with highest total pixel probability.
2. Reset selected neighbours and resort probabilities.
3. Propagate to neighbour with highest total pixel probability.
4. Reset selected neighbours and resort probabilities.
5. Propagate to neighbour with highest total pixel probability if the value is greater than 100.

As the crack propagates, the pixel that has been visited will be marked with a red colour on the bitmap grid describe in Section 4.4.2 above. A node on the graph is also created with a connection to the previous node.

For the first branch starting from the impact point, the neighbour with the highest total probability will be propagated to first. As shown in Figure 4.24 below, the first

neighbour selected is the one with total probability 245 pointed by the arrow. The pixel X indicates the start of the crack branch. Once the neighbour is selected, the coordinates are stored in the graph structure and the pixel is marked on a separate bitmap file.

230	201	114	0	10
245				0
221				5
101				50
98	79	150	170	165

Figure 4.24 Selection of first neighbour to propagate crack.

Since this is the first crack branch, a second neighbour has to be selected. Before the next selection is done, the probabilities of the selected neighbours as well as the neighbours around it will be reset to 0. By default, the two neighbours to the left and right of the selected neighbour will be reset to 0. The total probabilities of the second degree neighbours are also re-sorted. This is to prevent propagating cracks too near to each other. In Figure 4.25 below, the next neighbour to propagate the crack to will be the one with the highest total probability of 170.

Figure 4.25 Selection of second neighbour to propagate crack.

After the second selection is done, the neighbouring pixels will be reset to 0 and the probabilities re-sorted. A third crack branch will only be selected if the highest remaining total probability is at least 100. Figure 4.26 below shows that there will be a third crack branch since the highest total probabilities of the neighbours is 114.

Figure 4.26 Selection of third neighbour to propagate crack.

Each of the three crack branches will be stored in a queue and will be propagated in order of creation. For subsequent propagation of each crack branch, the condition to bifurcate the crack is shown below:

- *Highest total pixel probability value is greater than 100.* This condition is to allow bifurcation to occur at weaker areas of the material.
- *The crack is within a certain range from the start point.* This condition is checked by storing the distance the crack has travelled from the start point. A crack further away from the impact point should have a less chance to bifurcate. This is to simulate the loss of energy when a crack propagates.
- *The crack is a small distance away from a previous bifurcation point.* This condition is checked by storing the distance of the current crack from the start of its branch. This condition is necessary to prevent creating too many small fragments where a crack bifurcates and another newly bifurcated branch.

Bifurcation of subsequent crack branches occurs when the above conditions are satisfied. If it is not satisfied, the crack will only propagate to one of the neighbours with the highest total pixel probability calculated in step 1. When bifurcation occurs, a new crack branch is created and stored in a queue.

Step 3: Termination of a crack branch

At the end of every loop, the crack branch is checked for a terminating condition. The terminating condition is that when at least one or more of either the first degree or second degree neighbour has a total probability of -1. This is shown in the pseudo code in Figure 4.17 and Figure 4.18 above. This condition is satisfied when the current crack pixel is near the edge of the bitmap or near another crack branch. If the neighbouring pixels are not within the area of the bitmap, the coordinates of those

pixels are out of the range of the height and width of the bitmap. If the neighbouring pixels are from another crack branch, the pixels with similar coordinates on the bitmap grid are checked if they are being marked.

When a crack branch is terminated, the current node will be connected to the nearest boundary node or a branch node in the graph. Also, the connected node will be stored in a separate list for determining the number of broken pieces and the generation of broken fragments described in the next section.

4.5 3D Mesh Generation

4.5.1 Get boundary vertices of fragments and number of fragments

After the 2D crack propagation has completed, a crack pattern is formed as a closed directed graph. A list of terminating nodes from the graph is also stored. In Figure 4.27 below, a graph with terminated nodes indicated by boxes is shown. The number of broken fragments to be generated is equivalent to the number of terminating nodes. This is true for all cases where the crack branch will be terminated either at the edge of the bitmap or at another branch.

Figure 4.27 Graph with terminating nodes.

Once the number of fragments is known, the next step is to retrieve the boundary vertices of each fragment. This is done by traversing the graph and getting all non-overlapping cycles of the graph shown in Figure 4.28 below. The simplest way to do this is by starting from the terminating node and traversing a cycle in a counter-clockwise manner. While traversing, if a junction node is visited, the next node to traverse to will be the one on its left.

Figure 4.28 Graph traversal for boundary vertices.

Each set of boundary vertices for each fragment will be stored separately for processing. The boundary vertices are first processed to reduce its size. The size of the boundary vertices directly affects the number of triangles that will be rendered so reducing it will improve the performance. The best way to reduce the number while keeping the topology of the vertices is by skipping those vertices that are on the same line.

4.5.2 Triangulate boundary vertices

The boundary vertices are triangulated using a simple polygon triangulation method. This method involves the cutting of the ear of a polygon which is based on Meisters's theory [67]. An example of the simple polygon triangulation is shown in Figure 4.29 below.

Figure 4.29 A simple polygon triangulation.

The ear cutting method of triangulating simple polygons without holes is easy to implement. The source for this algorithm can be easily found for free [68]. Unlike other triangulation methods such as the constrained Delaunay triangulation algorithm [69-72], our method is fast and simple to implement.

4.5.3 3D extrusion and wrap-around vertices

To create a 3D mesh, the 2D polygon is extruded through its surface as shown in Figure 4.30 below. The extrusion depth depends on the depth of the original object. A total of three surfaces will be triangulated. The first two is the front and back cross-sectional surface. This is triangulated using the ear-cutting method with the back surface having the normals flipped. The shaded region in the figure shows the third surface that is the wrap-around mesh.

Figure 4.30 An extruded polygon.

The triangulation of the wrap-around surface is done by creating quads for every edge of the 2D polygon. This method of extrusion is fast and can be used for any polygon without holes.

Once the triangulation and extrusion is done, triangle strips are generated using fast and simple algorithms [73-75]. Each broken fragment mesh will be stored in one vertex buffer.

4.5.4 Physics properties

To animate the broken pieces using a physics engine, each newly created fragment have to be assigned some basic physical properties such as:

- Mass
- Friction
- Forces
- Center of mass

- Collision shape

Most of the physics properties will be re-used from the original object. However, the mass will be recalculated by taking the percentage of the fragment surface area to original surface area and use this percentage on the original object's mass. The center of mass also has to be re-calculated. Properties such as friction and restitution can be similar to the original object's settings.

To have a more realistic rigid body dynamics, a triangle list is used for the collision shape. Most fragments may have a concave shape or irregular shape and using a box as a collision will not make the fragments look real when animated. We use the same triangle list from Section 4.5.3 above to be the collision shape. Most physics engine supports the use of a triangle list as a collision shape.

4.6 Summary

This chapter presented the fracture framework that was implemented in our game engine. We showed how the probability texture and shape mask texture are used in the framework. Detailed information of each stage of the fracture pipeline was explained. The pipeline starts when an object is hit and ends after generating the broken pieces. The physics engine handles the rest of the simulation such as the rigid body dynamics and collision detection.

Chapter 5 Results

This chapter presents the results obtained from the fracture framework as images sequences and also a comparison tests to show the selection of crack propagation method and texture resolution. A performance chart is also presented to show the speed of the fracture framework in real-time.

5.1 Comparisons

This section shows the experiments made to select the method of crack propagation and the resolution of the textures used. The comparisons are based on the visual results and speed.

5.1.1 16-way and 8-way crack propagation comparison

The 2D crack propagation method discussed in the previous chapter is a 16-way crack propagation method. It involves propagating the crack to its 16 second-degree neighbouring pixels. We have also tried using an 8-way crack propagation method by only propagating the crack to its 8 first-degree neighbouring pixels.

For some cases, there may not be much difference between the 16-way and 8-way crack propagation. Figure 5.1 below shows the crack pattern generated from a 64x64 pixels probability texture with almost the same impact point near the center. Resolution and quality of both crack patterns look similar.

Figure 5.1 Comparison of 16-way (left) and 8-way (right) crack propagation.

However, the fundamental difference between the two methods is the degree of freedom. In the 16-way crack propagation, the crack can propagate to 16 different angles compared to just 8 with the 8-way method. This will result in a smoother and higher resolution crack pattern.

Figure 5.2 Crack patterns generated with an "S"-shaped probability texture.

In Figure 5.2 above a 64x64 pixel probability texture is used to generate the crack patterns. The 16-way crack pattern is shown in the center while the 8-way crack pattern is shown on the right. The crack pattern from the 16-way method is smoother around the curves and looks better than the 8-way method.

The main advantages of the 16-way crack propagation method compared to the 8-way method are listed in Table 5.1 below. The 16-way crack propagation method is chosen since it generates better-looking crack patterns.

Table 5.1 Comparison of features between the 16-way and the 8-way methods.

16-way Method	8-way Method
More freedom of propagation.	Only 8 directions. A diagonal line will always be at 45 degrees to the horizon.
Smooth curves.	Curves have jagged edges.
Fast. Create a node every 2 pixels since only the 2 nd degree neighbour is processed. This results in skipping a pixel at every propagation step hence reducing the number of nodes in the graph.	Creates a node at every crack propagation and propagates a pixel at a time. May create twice the number of nodes compared to the 16-way method.
More checking needs to be done. Both the 1 st and 2 nd degree neighbours need to be checked for termination.	Only check the 1 st degree neighbours needs to be checked and processed.

5.1.2 Pixel resolution of probability texture comparison

Probability textures of different resolutions are used to test the 2D crack propagation algorithm. The resolution of these textures will affect the visual quality of the crack patterns generated. For our test, a simple noise image for the probability texture and a circle shape mask is used. A set of different sizes is prepared. An example of the probability texture and mask texture is shown in Figure 5.3 below.

Figure 5.3 Probability and mask texture for resolution test.

Using the 2D crack propagation program, five 2D crack patterns are generated with the impact point near the center of the image. The textures of sizes, 16, 32, 64, 128 and 256 pixels are used. As described in the previous chapter, this 2D crack propagation program uses the 8-way method for the boundary mask and a 16-way method for the crack branches.

Figure 5.4 Crack pattern graph using resolution 16 (left) and 32 (right).

In Figure 5.4 above, the crack pattern generated from a low-resolution texture results in a crack pattern with low visual quality. The boundary shape of these two crack

patterns does not look like a circle as it shown in the shape mask texture in Figure 5.3 above.

Figure 5.5 Crack patterns using higher resolution textures.

Figure 5.5 above shows a crack pattern with texture resolution (a) 64x64 pixels, (b) 128x128 pixels and (c) 256x256 pixels. These crack patterns have a better visual quality than the ones in Figure 5.4. The shape boundary looks more like a circle and the crack branches look better. It can be seen that a higher resolution texture generates high-resolution crack patterns. More fragments can also be generated with higher resolution crack patterns. However, longer computation is needed to propagate the crack and more nodes will be created that would be very costly for higher resolution textures. A comparison of different texture sizes was done in our performance test presented in the next few sections.

5.2 Visual Results

This section shows image sequences to illustrate the motion of the broken pieces as well as the crack patterns itself. The image sequences are screenshots of the demo in a game engine taken a few milliseconds apart. Each column of the image sequences shows an instance of the object being fractured. This is to show the difference in the crack pattern if the object is hit at a different time or at a different area.

The objects are being fractured by a box thrown at it. The box is not breakable and is much harder than the objects being fractured.

5.2.1 Destruction of walls

The examples in this sub-section illustrate the fracture of different materials specified by different probability textures. Figure 5.8 to Figure 5.10 shows fracture of objects based on real materials such as glass concrete and wood. This shows how the probability texture is used to approximate realistic fracture. Figure 5.15 to Figure 5.17 shows fracture of objects with a custom material. This demonstrates the use of the fracture framework to break an object based on a custom texture.

The images in Figure 5.6 below shows the probability textures of glass, concrete and wood. The resolution for these textures is 128x128 pixels.

Figure 5.6 Probability textures of glass, concrete and wood.

The shattering of a glass wall in Figure 5.8 below shows crack lines that radiates from the impact point. Both image sequences show a fracture with almost the same impact point but with different crack patterns that was generated. The figure also illustrates that with a radial crack pattern, the shatter of glass looks similar to a real shattered glass in Figure 5.7 below.

Figure 5.7 Photo of shattered glass.

Figure 5.9 below illustrates a fracture of a concrete wall with crack lines defined in a probability texture in Figure 5.6 above. Both image sequences follows the crack lines of the probability texture to a certain extent based on the impact point. A noise background is used for the probability texture to have a more random effect.

Figure 5.10 below shows a wooden wall being broken through its grain lines. The broken fragments are long and thin since the probability texture shows horizontal grain lines.

Figure 5.8 Shattering a glass wall.

Figure 5.9 Breaking a concrete wall.

Figure 5.10 Breaking of a wooden wall.

The default probability texture is a grayscale noise effect as shown in Figure 4.6 above. Figure 5.14 below shows a wall breaking with a default probability texture. The crack pattern is generated by randomly propagating the crack away from the impact point.

Figure 5.15 below illustrates a wall being fractured with an “S”-shaped probability texture shown in Figure 5.11. Since the probability texture does not have the lines hitting the end of the texture, the crack will propagate with a straight line at random directions to the end of the texture when the crack propagates to the tip of the “S” shape. This is because the background of the probability texture is black. A noise background will make the crack propagate in random curve lines instead of a straight line.

Figure 5.11 A probability texture with an “S”.

The images in Figure 5.12 show that different instances of the wall breaking still results in an S-shaped crack pattern. The images are taken from the second sequence images of Figure 5.15 below.

Figure 5.12 The “S”-shaped crack pattern on a broken wall.

The probability textures in Figure 5.13 show a hexagonal grid texture on the left and a square grid texture on the right. These textures will generate crack patterns with custom crack edges. Figure 5.16 below shows a broken wall with jagged crack edges based on a hexagonal grid. Figure 5.17 below shows a broken wall with horizontal and vertical crack edges. A grid-like probability texture is best used to have random crack propagation while keeping a custom crack edge shape.

Figure 5.13 Probability textures with a hexagonal and square grid.

Figure 5.14 Broken wall with noise probability texture.

Figure 5.15 Wall with an “S”-shaped crack pattern.

Figure 5.16 Broken wall with a hexagonal crack pattern.

Figure 5.17 Broken wall with a square grid crack pattern.

5.2.2 Destruction of arbitrary shaped objects

The two examples in this sub-section show the use of the fracture framework to break thin objects with irregular cross-sectional shapes. The shape mask texture of these objects can be easily created in 3D Studio Max. The shape mask texture used in this sub-section is 128 by 128 pixels in size.

Figure 5.19 below illustrates a hand-shaped object being hit by a box at two different points. A default noise probability texture is used as the material. In the figure, a finger can be broken away from the hand in different ways. A shape mask texture of a hand is shown in Figure 5.18 below.

Figure 5.18 Shape mask of a hand (left) and heart (right).

Figure 5.20 below illustrates a heart-shaped object broken into pieces. The crack pattern is generated with a default noise probability texture. The shape mask texture used is shown in Figure 5.18 above.

Figure 5.19 A broken hand.

Figure 5.20 A broken heart.

5.2.3 House destruction

Objects in a game are not usually just thin extruded pieces with a square shape, unless they are walls or road signs. However, bigger objects can be made up of thin extruded objects, such as a house. A house can be made up of walls, doors, windows and a roof. This allows the fracture framework to create more complex objects by putting smaller objects together. The house model in Figure 5.21 below is made up of 7 smaller pieces.

Figure 5.21 Model of a house.

The four walls of the house have a concrete material while the roof is assigned a square grid probability texture shown in Figure 5.13 above. This is to show a compound object with different material. From the figure above, some of the walls have different shapes.

The shape masks of these walls are shown in Figure 5.22 below. The image on the left shows the shape of the wall in the front of the house where the black area is the door. The door and the wall at the front is two separate objects and can be assigned two separate materials. The image on the left is the shape of the walls on the sides of the house. The texture size used in this example is 64x64 pixels.

Figure 5.22 Shape mask of the walls of the house.

The door of the house is assigned a wooden material shown in Figure 5.6 above. Figure 5.23 below illustrates the destruction of a house at two different instances. Image sequence (a) shows the house being hit from the roof while image sequence (b) shows the house broken from the front wall.

By hitting a corner of the house the walls and the roof near the impact point will break first. The fragments created by the fracture of these walls will in turn create an impact on the other walls and roof causing them to fracture next. The image sequences also shows the house being broken into fragments of different shapes and sizes and looks different every instance it breaks.

Figure 5.23 Destruction of a house.

5.2.4 Application in a game environment

The previous sections show the results of the fracture framework in a simple game engine environment. Most effects were turned off and only the fracture objects and the ground are put into the scene. Only the physics and texture rendering are turned on

This section shows the results in an actual game environment. This environment has the trees, wildlife, sky, particle effects on vehicles and artificial intelligence of non-player vehicles turned on.

Figure 5.24 Screenshots in a game.

Figure 5.24 shows the breaking of a few concrete walls placed on a racing track. Each wall is modeled with six quad surfaces. The rally cars were destroying the walls as

they race. Although a racing game is one example of presenting an applied result, this framework can be used in any game genre such as real-time strategy or first person shooter where a player will like to destroy objects and see them broken differently each time. In this example, the game runs in about 50 FPS.

5.3 Performance

The fracture framework was coded in C++ using Microsoft Visual Studio 2005. It was tested on the proprietary engine used by TQ Global. The physics engine from TQ Global library was used to drive the animation of the fractured objects. The specification of our test machine to get the performance results of the projects is shown in the Table 5.2 below.

Table 5.2 Specification of test machine

CPU	2.66 GHz Intel Core 2
GPU	NVIDIA GeForce 8600 GTS with 256 MB RAM
Memory	2 GB RAM
Operating System	Microsoft Windows XP Professional with DirectX 9

The performance of the fracture framework depends on two factors; i.e. the resolution of the probability and shape mask textures and the number of fragments generated. Firstly, a higher resolution texture will generate better-looking fragments as shown in the previous sections. However, the crack propagation method will take a long time to generate the crack pattern for these high-resolution textures due to the number of pixels to be processed. Secondly, if more fragments are generated, more time is needed to generate the mesh and its physical properties. A high-resolution texture also affects the number of fragments created since the crack has more areas to propagate to and have more chances to bifurcate.

Table 5.3 Performance table for an object with a texture size of 64x64 pixels.

Number of Fragments	2D Crack Propagation (ms)	3D Mesh Generation (ms)	Fragment Creation (ms)	Total Time (ms)
5	0.75	1.91	2.12	4.85
8	1.02	2.19	3.10	6.38
11	1.27	3.36	5.07	9.78
15	1.44	3.57	5.22	10.30
20	1.72	4.89	6.70	13.39

Table 5.3 shows the timings for breaking a wall similar to Figure 5.14 above but with a probability and shape mask texture of 64 by 64 pixels. This texture size is the best size to be used in the framework as it can create high resolution crack patterns similar to the 128 and 256 pixel ones and is fast enough to be run in real-time without having noticeable drop in frame rates. The table shows that with 20 fragments generated, the total fracture process takes less than 15 ms to complete. Since a frame of a game is usually processed in less than 30 ms, this should not affect the quality of the gameplay. The table also shows that the time taken for the fracture of an object is linearly proportional to the number of fragments that will be created.

The total time taken for each object listed in Table 5.3 above includes all the stages in the fracture framework process described in Figure 4.1 above. While the timing for the fracture initiation process is not listed, every instance of a fracture takes about 0.1 ms just to read in the pixel value at the impact point. This time is the same for any texture size or object. It can also be seen that the fragment creation process takes the longest time for each fracture and it takes almost half of the total time taken. The reason it takes a long time is due to the creation of new models. This involves filling the vertex buffers with vertices of the new fragments and creating the collision triangle list of the rigid body. This process takes a longer time for more fragments that will be created.

Table 5.4 Performance table for a wall broken with varying texture sizes.

Texture Size	Number of Fragments	2D Crack Propagation (ms)	3D Mesh Generation (ms)	Fragment Creation (ms)	Total Time (ms)
16	3	0.19	0.42	0.44	1.12
32	8	0.54	1.44	1.98	4.03
64	15	1.44	3.57	5.22	10.30
128	15	2.84	11.25	16.56	30.72
256	25	8.94	24.52	68.44	102.06

Table 5.4 shows the breakdown of timings for fracturing a wall using different sizes of probability and shape mask textures. For texture size of 16 and 32 pixels, a maximum of 3 and 8 fragments respectively can be generated. The other texture sizes allow more fragments to be generated. In this table, using a 64-pixel texture will be able to generate as many fragments as the other bigger textures.

The results of breaking a square wall shown in Figure 5.8 to Figure 5.10 and Figure 5.14 to Figure 5.17 above have a total fracture time of about 20 to 30 milliseconds. A 128 by 128 pixels texture is used and the walls are broken in 10 to 20 fragments with various materials each.

Figure 5.19 and Figure 5.20 above shows an irregular shaped objects being broken. These objects take roughly 3 milliseconds more for the fracture process compared to a square wall with the same texture size and at least 10 fragments. This additional time is caused by the algorithm to create the boundary vertices of the original object from the shape masks.

The house in Figure 5.23 above is broken into about 60 fragments in total from 7 combined objects. The total fracture time for the house is 45 milliseconds. Although

the texture size used is 64x64 pixels, 7 objects are fractures almost at the same frame. This causes the fracture of the whole house to take some time.

The performance of fracturing objects in games also depends on how many objects are broken at the same time. If more objects have to be broken at the same time, a smaller size texture should be used but at the expense of the crack pattern resolution. If the fracture time exceeds 30 milliseconds, there will be some lag or sudden increase of frame rate in the game when the object is hit. This is caused by the CPU waiting for the fracture algorithm to complete.

5.4 Summary

In this chapter, we first discussed why the 16-way propagation method was chosen over the 8-way one. Then, results of fracture based on different materials were shown by using different probability textures. Fracture of arbitrary shaped objects was shown by using different shape mask textures. More complex fracture was shown by breaking a compound object such as a house. To show that our method can be implemented in games, we showed screenshots of wall breaking by cars in a game environment. Lastly, our performance shows that the method proposed can be run in real-time.

Chapter 6 Conclusions

This chapter provides a summary of the research in this thesis. The last section presents ideas for the next step in this research.

6.1 Summary

This thesis has shown that the goal of dynamically fracturing an object in real-time is met. We proved this by implementing a fracture framework that can be fitted into a game engine with a physics engine that can generate new broken fragments in run-time. We achieve a fracture time of 10 milliseconds or less in order to achieve frame rates of at least 30 FPS.

Our method allows us to fracture objects that have low triangles count, such as a simple surface mesh of a box. This allow us to load many objects that can be fractured and reduce the loading time of the game. New high polygonal meshes of the broken fragments are then generated when the objects are hit and fractured. The broken fragments can be removed after some time and be replaced by the original low polygonal mesh. This will also save memory usage of the game.

The results shown in Chapter 5 illustrates that the fracture of objects dynamically can be done in real-time. This is an important specification for a feature to be run during a game. To achieve that speed, we have to compromise the accuracy by using a procedural method instead of a physical approach. However, we can still approximate a physical fracture by using probability textures that have similar patterns to a real-life crack.

Another advantage to using textures to generate crack patterns is that there is flexibility in creating a custom, image-based crack pattern. The user can create different probability textures to break objects differently instead of remodeling an object to break it in a different way. The objects presented with this fracture framework can be modeled with low polygon count but the fragments created will be of higher polygon count depending on the size of textures used. This will also save on memory usage for breakable objects in the game compared to a tetrahedralized object.

6.2 Future Work

Our technique is limited to objects that have a 2D cross-sectional surface which are extruded. We considered fracturing only solid objects as shown in Section 1.3. When solid objects are fractured, the broken pieces generated are also solid fragments. This prevents most 3D objects to be fractured using our technique. For example, cylindrical objects cannot be fractured using this technique. A 2D crack pattern texture can be mapped onto a 3D cylindrical object but this 2D crack pattern could not be used to create solid fragments using our algorithm. Only surface fragments can be generated from that. To solve this, a 3D crack propagation method is needed. A suggestion would be to make use of the cube or volume texture as the probability texture. This will give a more accurate material structure of the object. The crack will then propagate from the impacted surface through to the other end of the object. This method is also a good way to approximate a finite element method. To simplify this method, a planar crack could be used instead of a line crack to reduce computation of re-triangulation of the internal vertices of the mesh.

To improve the visual appearance, some particle effects can be used when the object is hit. For example, when a concrete wall is hit some small concrete bits can be

emitted to simulate very small broken fragments. This feature is part of most physics engine. 2D textures of small particles are rendered and animated using the rigid body dynamics of the physics engine. This method gives a nice effect that is fast and does not use much computing resources compared to modeling small fragment meshes.

The above two suggestions will improve the visual appearance of a fracture and solve the limitations of the current method. It is best to improve on the visual appearance since gamers nowadays have more demand on the effects of destruction in games. This can be easily achieved with the use of next-generation gaming hardware.

Bibliography

- [1] D. Baraff and A. Witkin, "Physically Based Modeling: Principles and Practice, Differential Equation Basics," *SIGGRAPH '97 Course Notes*, 1997.
- [2] A. Shukla, *Dynamic Fracture Mechanics*. World Scientific Publishing, 2006.
- [3] L. B. Freund, *Dynamic Fracture Mechanics*. Cambridge University Press, 1998.
- [4] B. R. Lawn, *Fracture of Brittle Solids*, 2nd ed. Cambridge University Press, 1993.
- [5] T. L. Anderson, *Fracture Mechanics: Fundamentals and Applications*, 3rd ed. CRC Press, 2005.
- [6] B. Steverding and S. H. Lehnigk, "Dynamics of Brittle Fracture," *International Journal of Fracture*, vol. 8, pp. 59-73, 1972.
- [7] D. Gross and T. Seelig, *Fracture Mechanics: With an Introduction to Micromechanics*. Springer, 2006.
- [8] A. S. Jayatilaka, *Fracture of Engineering Brittle Materials*. Applied Science Publishers, 1979.
- [9] M. Neff and E. Fiume, "A Visual Model for Blast Waves and Fracture," *Proceedings of the 1999 conference on Graphics Interface '99*, pp. 193-202, 1999.
- [10] W. D. Callister Jr, *Materials Science and Engineering: An Introduction*, 4th ed. John Wiley & Sons, Inc., 1991.
- [11] A. M. Howatson, P. G. Lund, and J. D. Todd, *Engineering Tables and Data*. Chapman & Hall, 1972.
- [12] A. Ivankovic and M. H. Aliabadi, *Crack Dynamics*. WIT Press, 2005.

Bibliography

- [13] F. Abraham, "Dynamics of Brittle Fracture with Variable Elasticity," *Physical Review Letters*, vol. 77, pp. 869-872, 1996.
- [14] C. Hecker, "Physics, the Next Frontier," in *Game Developer*, October/November 1996, pp. 12-20.
- [15] P. Sweeney, A. Norton, R. Bacon, D. Haumann, and G. Turk, "Modelling Physical Objects for Simulation," in *Proceedings of the 23rd conference on Winter Simulation* Phoenix, Arizona, United States: IEEE Computer Society, 1991.
- [16] D. H. Eberly, *Game Physics*. Morgan Kaufmann, 2004.
- [17] C. Hecker, "Physics, Part 2: Angular Effects," in *Game Developer*, December 1996/January 1997, pp. 14-22.
- [18] C. Hecker, "Physics, Part 4: The Third Dimension," in *Game Developer*, June 1997, pp. 15-25.
- [19] C. Hecker, "Physics, Part 3: Collision Response," in *Game Developer*, March 1997, pp. 11-18.
- [20] D. Baraff and A. Witkin, "Physically Based Modeling: Principles and Practice, Rigid Body Dynamics," *SIGGRAPH '97 Course Notes*, 1997.
- [21] E. C. Prakash, *3d Computer Game Programming*, 2nd ed. Addison Wesley, 2005.
- [22] C. Ericson, *Real-Time Collision Detection*. Morgan Kaufmann, 2005.
- [23] J. O'Rourke, *Computational Geometry in C*, 2nd ed. Cambridge University Press, 1998.
- [24] M. Moore and J. Wilhelms, "Collision Detection and Response for Computer Animation," *Computer Graphics*, vol. 22, 1988.
- [25] J. W. Simons, S. W. Kirkpatrick, R. W. Klopp, and L. Seaman, "Methods for Modeling Damage in Finite Element Calculations," in *Proceedings of International Seminar on Quasi-Impulsive Analysis* Osaka, Japan, 1999.

Bibliography

- [26] K. H. Huebner, E. A. Thornton, and T. G. Byrom, *The Finite Element Method for Engineers*, 3rd ed. John Wiley, 1995.
- [27] B. Cutler, J. Dorsey, L. McMillan, M. Müller, and R. Jagnow, "A Procedural Approach to Authoring Solid Models," *ACM Transactions on Graphics*, vol. 21, pp. 302-311, 2002.
- [28] B. Zhaosheng, H. Jeong-Mo, T. Joseph, and F. Ronald, "Fracturing Rigid Materials," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 370-378, 2007.
- [29] J. F. O'Brien and J. K. Hodgins, "Animating Fracture," *Communications of the ACM*, vol. 43, pp. 68-75, 2000.
- [30] J. F. O'Brien and J. K. Hodgins, "Graphical Modeling and Animation of Brittle Fracture," *Proceedings of the 26th annual conference on Computer Graphics and Interactive Techniques*, pp. 137-146, 1999.
- [31] J. F. O'Brien, A. W. Bargteil, and J. K. Hodgins, "Graphical Modeling and Animation of Ductile Fracture," *Proceedings of the 29th annual conference on Computer Graphics and Interactive Techniques*, pp. 291-294, 2002.
- [32] J. F. O'Brien, "Graphical Modeling and Animation of Fracture," PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, 2000.
- [33] M. Muller, M. Teschner, and M. Gross, "Physically-Based Simulation of Objects Represented by Surface Meshes," *Computer Graphics International*, pp. 26-33, 2004.
- [34] M. Müller, L. McMillan, J. Dorsey, and R. Jagnow, "Real-Time Simulation of Deformation and Fracture of Stiff Materials," *Proceedings of the Eurographic workshop on Computer Animation and Simulation*, pp. 113-124, 2001.
- [35] D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture," *Proceedings of the 15th annual conference on Computer Graphics and Interactive Techniques*, pp. 269-278, 1988.

Bibliography

- [36] A. Norton, G. Turk, B. Bacon, J. Gerth, and P. Sweeney, "Animation of Fracture by Physical Modeling," *The Visual Computer: International Journal of Computer Graphics*, vol. 7, pp. 210-219, 1991.
- [37] K. Hirota, Y. Tanoue, and T. Kaneko, "Simulation of Three-Dimensional Cracks," *The Visual Computer*, vol. 16, pp. 371-378, 2000.
- [38] D. Steinemann, "Generation and Fracturing of Thick Shells," in *Central European Seminar on Computer Graphics*, 2005.
- [39] J. Smith, A. Witkin, and D. Baraff, "Fast and Controllable Simulation of the Shattering of Brittle Objects," *Computer Graphics Forum*, vol. 20, pp. 81-91, 2001.
- [40] O. Mazarak, C. Martins, and J. Amanatides, "Animating Exploding Objects," *Proceedings of the 1999 conference on Graphics Interface '99*, pp. 211-218, 1999.
- [41] C. Martins, J. Buchanan, and J. Amanatides, "Visually Believable Explosions in Real Time," *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*, pp. 237-259, 2001.
- [42] G. Yngve, J. O'Brien, and J. Hodgins, "Animating Explosions," in *Proceedings of the 27th annual conference on Computer Graphics and Interactive Techniques*, 2000, pp. 29-36.
- [43] G. Taubman and E. Chang, "A Fast Fracture Method for Exploding Structures," *International Conference on Computer Graphics and Interactive Techniques*, 2004.
- [44] L. A. Ek, R. Vistnes, and O. E. Gundersen, "Animating Physically Based Explosions in Real-Time," in *Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* Grahamstown, South Africa: ACM, 2007.
- [45] S. Gobron and N. Chiba, "Crack Pattern Simulation Based on 3d Surface Cellular Automata," *The Visual Computer*, vol. 17, pp. 287-309, 2001.

Bibliography

- [46] S. Gobron and N. Chiba, "Visual Simulation of Crack Pattern Based on 3d Surface Cellular Automaton," in *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*: IEEE Computer Society, 2000.
- [47] H. N. Iben and J. F. O'Brien, "Generating Surface Crack Patterns," *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 177-185, 2006.
- [48] K. Hirota, Y. Tanoue, and T. Kaneko, "Generation of Crack Patterns with a Physical Model," *The Visual Computer*, vol. 14, pp. 126-137, 1998.
- [49] P. Federl and P. Prusinkiewicz, "Finite Element Model of Fracture Formation on Growing Surfaces," in *Proceeding of Computational Science, ICCS 2004: 4th International Conference*, 2004.
- [50] N. Zhang, X. Zhou, D. Sha, X. Yuan, K. Tamma, and B. Chen, "Integrating Mesh and Meshfree Methods for Physics-Based Fracture and Debris Cloud Simulation," 2006.
- [51] M. Pauly, R. Keiser, B. Adams, P. Dutré, M. Gross, and L. J. Guibas, "Meshless Animation of Fracturing Solids," *International Conference on Computer Graphics and Interactive Techniques*, pp. 957-964, 2005.
- [52] M. Müller, B. Heidelberger, M. Teschner, and M. Gross, "Meshless Deformations Based on Shape Matching," *Proceedings of ACM SIGGRAPH 2005*, vol. 24, pp. 471-478, 2005.
- [53] A. Martinet, E. Galin, B. Desbenoit, and S. Akkouche, "Procedural Modeling of Cracks and Fractures," *Proceedings Shape Modeling Applications*, pp. 346-349, 2004.
- [54] B. Desbenoit, E. Galin, and S. Akkouche, "Modeling Cracks and Fractures," *The Visual Computer*, vol. 21, pp. 717-726, 2005.
- [55] R. Allègre, A. Barbier, E. Galin, and S. Akkouche, "A Hybrid Shape Representation for Free-Form Modeling," *Shape Modeling Applications, 2004. Proceedings*, pp. 7-18, 2004.

Bibliography

- [56] R. Allègre, E. Galin, R. Chaine, and S. Akkouche, "The Hybridtree: Mixing Skeletal Implicit Surfaces, Triangle Meshes, and Point Sets in a Free-Form Modeling System," *Graphical Models*, vol. 68, pp. 42-64, 2006.
- [57] D. Mould, "Image-Guided Fracture," *Proceedings of the 2005 conference on Graphics Interface*, pp. 219-226, 2005.
- [58] B. Wyvill, K. v. Overveld, and S. Carpendale, "Rendering Cracks in Batik," in *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering* Annecy, France: ACM, 2004.
- [59] M. Gavrilova and M. Alsuwaiyel, "Two Algorithms for Computing the Euclidean Distance Transform," *International Journal of Image and Graphics*, vol. 1, pp. 635-645, 2001.
- [60] S. Raghavachary, "Fracture Generation on Polygonal Meshes Using Voronoi Polygons," in *ACM SIGGRAPH 2002 Conference Abstracts and Applications* San Antonio, Texas: ACM Press, 2002.
- [61] D. Eberle, O. S. Havok, and R. O'Sullivan, "A Procedural Approach to Modeling Impact Damage," *Proceedings of the SIGGRAPH 2003 Conference on Sketches & Applications: in conjunction with the 30th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 1-1, 2003.
- [62] S. Workman, "A Cracking Algorithm for Destructible 3d Objects," M.S. thesis, University of Sheffield, 2006.
- [63] A. Keith, "Demolishing Buildings in Computer Games," University of Sheffield, 2002.
- [64] D. Fox, "Demolishing Objects in Computer Games," University of Sheffield, 2003.
- [65] O. Oda and S. Chenney, "Fast Dynamic Fracture of Brittle Objects," in *ACM SIGGRAPH 2005 Posters* Los Angeles, California: ACM Press, 2005.
- [66] O. Oda and N. Subramaniam, "Fast Dynamic Fracture of Brittle Objects in 3d," in *ACM SIGGRAPH 2006 Research posters* Boston, Massachusetts: ACM Press, 2006.

Bibliography

- [67] G. H. Meisters, "Polygons Have Ears," *American Mathematical Monthly*, vol. 82, pp. 648-651, 1975.
- [68] I. Garton, "Ear Cutting for Simple Polygons," [Online]. Available: http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/lan/cutting_ears.html. [Accessed Jan 30, 2008].
- [69] T. P. Fang and L. A. Piegl, "Algorithm for Constrained Delaunay Triangulation," *The Visual Computer: International Journal of Computer Graphics*, vol. 10, pp. 255-265, 1994.
- [70] L. Paul Chew, "Constrained Delaunay Triangulations," *Algorithmica*, vol. 4, pp. 97-108, 1989.
- [71] S. W. Sloan, *A Fast Algorithm for Generating Constrained Delaunay Triangulations*. University of Newcastle, 1991.
- [72] R. Klein, "Construction of the Constrained Delaunay Triangulation of a Polygonal Domain," *CAD Systems Development*, pp. 313-326, 1995.
- [73] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola, "Constrained Strip Generation and Management for Efficient Interactive 3d Rendering," *Computer Graphics International 2005*, pp. 115-121, 2005.
- [74] F. Evans, S. Skiena, and A. Varshney, "Optimizing Triangle Strips for Fast Rendering," *Proceedings of the 7th Conference on Visualization '96*, pp. 319-326, 1996.
- [75] L. Velho, L. H. de Figueiredo, and J. Gomes, "Hierarchical Generalized Triangle Strips," *The Visual Computer*, vol. 15, pp. 21-35, 1999.