Procedural Destruction of Objects for Computer Games

PUBLIC VERSION

THESIS

submitted in partial fulfilment of the

requirements for the degree of

MASTER OF SCIENCE

in

MEDIA AND KNOWLEDGE ENGINEERING

by

Joris van Gestel

born in Zevenhuizen, the Netherlands

Computer Graphics and CAD/CAM Group
Department of Mediamatics
Faculty of EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cannibal Game Studios
Molengraaffsingel 12-14
2629 JD Delft
The Netherlands
www.cannibalgamestudios.com

Author: Joris van Gestel
Student id: 1099825
Email: j.vangestel@cannibalgamestudios.com
Date: May 10, 2011

Summary

Traditional content creation for computer games is a costly process. In particular, current techniques for authoring destructible behaviour are labour intensive and often limited to a single object basis. We aim to create an intuitive approach which allows designers to visually define destructible behaviour for objects in a reusable manner, which can then be applied in real-time.

First we present a short introduction into the way that destruction has been done in games for many years. To better understand the physical processes that are being replicated, we present some information on how destruction works in the real world, and the high level approaches that have developed to simulate these processes.

Using criteria gathered from industry professionals, we survey previous research work and determine their usability in a game development context. The approach which suits these criteria best is then selected as the basis for the approach presented in this work. By examining commercial solutions the shortcomings of existing technologies are determined to establish a solution direction.

To separate destructible behaviour from particular objects, we introduce the concept of destructible materials: where the material of an object usually defines the way an object looks, a destructible material will determine how it breaks. Destructible materials provide a reusable definition and intuitive way of designing and tweaking destructible behaviour of objects, which can then be applied in real-time.

Using a prototype implementation we show the viability of the presented approach and how it extends previous research with reusability, making it more designer friendly and allowing the same destructible behaviour to be easily applied to different objects. While the prototype can only apply this destructible behaviour in real-time for simple cases, it still takes us a step in the right direction.

Acknowledgements

I would like to thank my colleagues for all their continued efforts to keep me motivated, for their feedback and insights, and for creating the supporting and inspiring environment in which I could work on this project, and finally see it completed.

I would also like to thank my supervisor, Rafael Bidarra, for his time, his patience and his guidance, and his efforts to have me do this project to the best of my abilities.

Finally, I would also like to thank my friends and family for their love and support, and pretending to understand what I was talking about during my many rants.

While I hardly ever express it directly, or in this case with many words, I can ensure you all, it is not less heartfelt.

Thank you.

Joris van Gestel

Zevenhuizen, the Netherlands

Monday, 11 April 2011

Contents

1 Introduction

With the increase in gaming console and personal computer processing power, the demand for higher quality graphics as well as higher quality gameplay increase alongside it. Higher quality in these cases means more work as more content is required to fill ever bigger game worlds, but at the same time this content also needs to be more detailed.

Creating content for games has always been, and remains so largely today, created by hand, particularly adding destructibility to a gaming environment. This makes content creation a very costly process, costs which get exponentially higher as the demand for more, and more detailed, content increases. A lot of research has been performed, and effort expended, in the last few years trying to automate this process by developing procedures that are capable of automatically creating content, like for instance virtual worlds. As making content destructible takes up a very large portion of the total time needed to create content, this would be an excellent area to apply this automation to as well.

Goal

The goal of this master thesis project is to come up with a procedural approach to automatically destroy objects. The behaviour and visual effects created by this procedure will need to be designable by artists, and the algorithms used need to perform fast enough to run in real time.

Background

To better understand why content creation is such a timely process, how the physical processes of destruction and the different approaches to simulating these processes work, some background information on these subjects is first discussed in chapter 2.

A lot of research has already been performed into techniques for simulating destruction. A selection of these techniques has been examined to determine their strengths and weaknesses. To see if any of them could be used as a basis for our new approach, all approaches are compared through a set of criteria. The results of this survey can be found in chapter 3.

To achieve more interactivity and destructibility in games, several gaming companies and middleware developers have already created technologies for procedural destruction, but these technologies are not yet used in a lot of games. By examining these techniques closely and using them where possible, some insight can be gained into why these technologies are not widely used in current games. The findings of this study can be found in chapter 4.

Approach & Prototype

In chapter 5, a new approach will be presented which will present a novel concept with regards to automated destruction of objects for games. Several new ideas will be presented as to how a designer could go about generically designing destructible behaviour in an intuitive way while maintaining sufficient control to ensure constraints placed on the gaming environment are not violated.

Chapter 6 will contain the description of a prototype implementation of the presented concepts and ideas. While this prototype will only contain a subsection of the presented possibilities it will demonstrate the feasibility of the approach and create a basis for future work.

A high level description of the architecture of the prototype and the developed algorithms is given in chapter 7. This will provide an overview of the structure of the prototype and explain how certain encountered problems were solved.

In chapter 8 the important implementation details of the procedures are discussed, as well as the specifics of different solution directions which were explored but which were found to not properly or adequately solve encountered problems.

Results

In chapter 9 the results achieved by this project are discussed using a showcase. This showcase demonstrates how the approach described in this thesis can be used and what the results of applying the approach looks like.

Finally chapter 10 contains conclusions on the feasibility of the presented approach, future work, and recommendations to Cannibal Game Studios.

2 Background

In this chapter, some background information into the subject of destruction in different contexts is discussed. In section 2.1, an overview of the traditional way of creating destruction in computer games is given. In section 2.2, some basic information on the physics behind destruction is presented to allow us to understand the real world behaviour we are trying to simulate. Finally in section 2.3, an overview is presented of the different approaches that have been developed for this simulation of the real world behaviour of destruction.

2.1 Destruction in games

Creating a piece of content is a multi-step process, see Figure 2.1. Once a particular piece of content is needed, concept art is usually created first to determine how the content should look, based on the art direction of the project and the purpose of the content. This concept art can for example consist of artistic drawings of the piece of content in its natural environment, or contain more technical sketches of different variations or configurations, for instance a car with its door opened and closed.

Figure 2.1 The process of creating content.

Based on the concept art, a three dimensional model is then created. For most current generation games, two models are actually needed; a low-detail in-game model and a high-detail model. Most games use techniques like normal mapping to simulate high surface detail without requiring the actual in-game model to be highly detailed. The high detail model is used to generate a projection onto the in-game model, the in-game model and this projection can then be used at runtime to quickly render what appears to be a high-detail model.

Once the in-game model has been completed it needs to be UV-mapped, which is the process of laying out all polygons contained in the mesh onto a flat two dimensional surface (in which the axis are named U and V, instead of X and Y). This creates the effect of painting

an image onto the surface of the three dimensional model, called a texture. Once a texture has been created the content is ready to be used in a game.

Traditionally content has always been static once it is placed in a game world. It might be moved around, scaled or rotated, but no changes could be made to the model or texture at runtime, as this was too demanding on system resources. However, at times, changes to a piece of content are desired, for instance when an object receives damage. Consider a car and suppose a collision occurs with another object, as a result, parts of the car might become dented and paint gets damaged. Changes like this could not be made, as it would require the dented surface to become more detailed adding, changing or removing polygons in the model. As the layout of the model changes, the UV mapping needs to be updated as well, to ensure the entire surface of the model still maps to the texture correctly. The texture itself would also have to be adapted to simulate the loss of paint from the collision or other inflicted damage.

To work around this limitation, once a change is needed to a piece of content, its model is simply removed from the scene and a new model, which looks largely the same, is placed at the exact same position. As this switch happens in between frames the effect is not noticeable by the user; from his perspective the performed action simply gave the expected result (his car is now dented). This approach is therefore known as art swapping, see Figure 2.2.

Figure 2.2 Traditional art swap method. Two independent and completely different collisions result in the exact same animation being displayed. Images from **(1)**.

One of the downsides of this approach is that the result will always be exactly the same; the dent in the car will always be in the exact same location, regardless of where the actual impact took place. To make this approach more convincing, modern games often use several different, progressively more damaged versions of their content which are swapped based on the situation at hand, to make the animation more accurate. However, this is also the second downside of this approach, steps three, four and five in Figure 2.1 now have to be

repeated for every variation of the piece of content, adding considerably to the amount of time needed to create one piece of content.

2.2 The physics of destruction

In traditional material sciences, deformation is defined as a change in the shape or size of an object as a force is applied to it. These forces are one of four types, tensile (pulling), compressive (pushing), shear (sliding) and bending or torsion (twisting). Deformations are often described in terms of strain and are the result of stresses internal to an object, which form as forces are applied to the object, or as its temperature changes. This strain is a geometrical measure of the deformation, representing the relative displacement between particles in the material body.

There are different kinds of deformations, namely elastic, plastic and fracture. Elastic deformations are reversible: as soon as the applied force is removed the deformed object returns to its original shape. For instance consider a piece of sheet metal; if a force is applied to one end while the other is held in place, it will bend slightly but it will return to its original shape as soon as this force is removed.

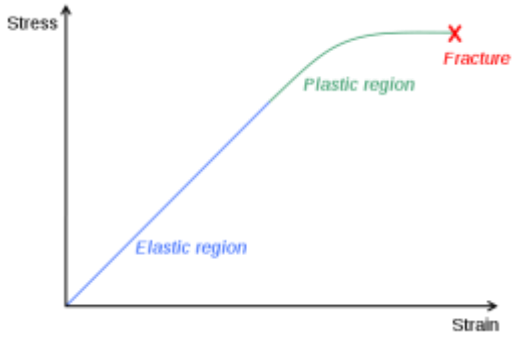


Figure 2.3 Relationship between stress (force) and strain (deformation) of a ductile material. Image by **(28)**.

Plastic deformations, on the other hand, are not reversible. When an object undergoes plastic deformation, it no longer returns to its original shape once the applied force is removed. However, before an object can undergo plastic deformation, it will have undergone elastic deformation first, even if only by a very small amount. Consider again the piece of sheet metal: if a larger force than last time is applied, bending it further, it will not return to its original shape once the force is removed.

Fractures are also a type of deformation, and like plastic deformations they are not reversible. Fractures occur after an object has reached the end of its elastic, and then plastic, deformation ranges (see Figure 2.3). Therefore, if enough force is applied, any object will fracture. Consider yet again the piece of sheet metal: if twisted a small amount, it will return to its original form, twisted further it will keep its new form, but if twisted even further beyond this point, cracks will appear until finally you are left with two separate pieces.

As there are different kinds of deformation, so are there different kinds of materials. Ductile materials are able to deform plastically without fracture, for instance clay. Brittle materials on the other hand, appear to fracture before deforming elastically, like glass. These materials do actually deform elastically and plastically, but only by a very small amount, so as soon as more force is applied than can be absorbed, they appear to directly go from a non-deformed state to a fractured state.

2.3 Simulating destruction

Various different methods and techniques exist for simulating objects deforming and fracturing, some of them based on replicating the actual physics of how these processes take place, as described in the previous section, and some of them based on approximations or simplifications instead. A few distinctions can be made between the various methods, see (2) for an in depth report. On the highest level, we can consider the distinction between Lagrangian and Eulerian methods:

Lagrangian. The model consists of a set of points, with varying locations. Each point stores the material properties of the model at that point.

Eulerian. Model properties are computed for a set of stationary grid points which store the material properties at those points and how they change over time.

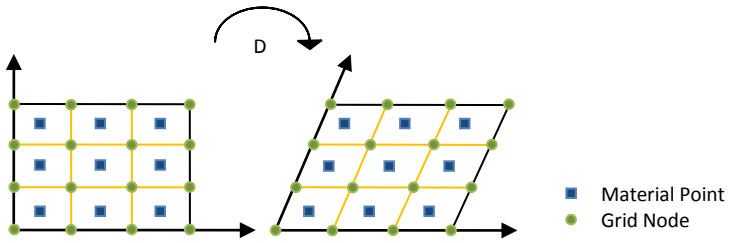


Figure 2.4 A Lagrangian mesh, the mesh deforms with the material body.

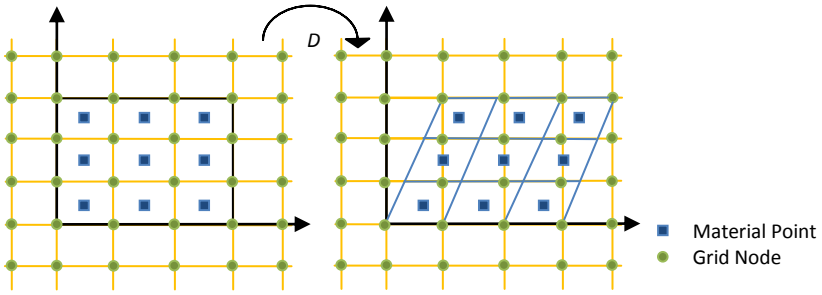


Figure 2.5 An Eulerian mesh, the material body flows through the mesh as it deforms.

Lagrangian methods are therefore usually the chosen way of representing models in rigid and soft body simulations, due to the convenience of being able to express the model as a connected mesh or a cloud of points, in the same way as the visual model is expressed. Eulerian methods on the other hand are the preferred way of simulating fluids because of the simpler formulas involved. Also in Eulerian methods boundaries of objects are not explicitly defined, and it is very difficult to use them for anything other than simulating fluid dynamics, as most other objects are described by a surface boundary representation, e.g. the visible polygonal shell of the object. As a result, Eulerian methods are not of interest for the subject of this project and we only focus on Lagrangian methods.

Even for Lagrangian methods, the models used vary a lot regarding specific methods and implementations. The type of ***model*** used can usually be divided into two categories:

Meshed. Relies on a grid or a mesh of interconnected simulation points.

Mesh free. All simulation points are independent of each other.

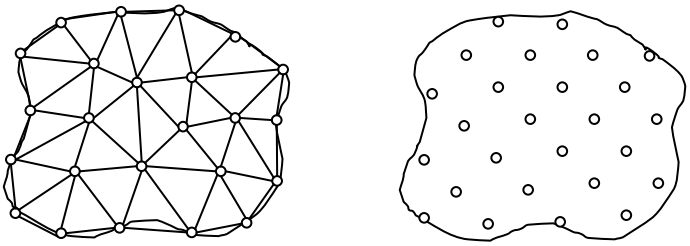


Figure 2.6 On the left a meshed model, all nodes in the model are interconnected. On the right a meshless model, no connections are present between the nodes.

Traditional physically-based simulation methods usually use a meshed model, as this is very close to the actual physicality of materials, where atoms and molecules are also all interconnected with neighbouring particles. Meshfree methods, due to their lack of interconnectivity between particles, are therefore usually geometric-based methods, as they lack the means to locally specify material properties.

Besides the various way of describing the models used, another important characteristic is the way time integration is handled. While the time integration is independent of the simulation method, it still plays an important part in the overall stability of the solution. The two main variations of ***time integration*** are:

Explicit Euler, also known as forward Euler.

Implicit Euler, also known as backwards Euler.

In the explicit method, Newton's second law of motion can be described by the two following formulas:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \mathbf{v}(t)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t F(\mathbf{v}(t), \mathbf{x}(t), t)$$

This method is called explicit because it provides explicit formulas for the values at the next time step. While explicit methods are easy to implement, they can destabilize easily due to them extrapolating a constant right hand side blindly into the future. For large values of Δt this can lead to overshooting of goal or equilibrium positions (see Figure 2.7), requiring even larger forces to return the system within its boundary conditions.

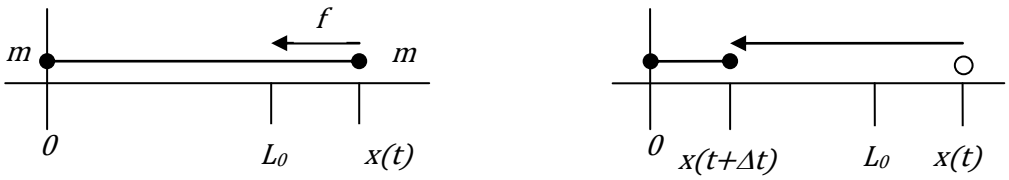


Figure 2.7 A spring is fixed at the origin, with a free point with mass *m* at *x(t)*. Force *f* pulls the free point to the equilibrium location L_0, however, for large values of Δt the point overshoots L_0 and energy is erroneously added to the system.

This can lead to an exponential increase in energy contained in the system, which will then lead to explosion of the entire system. In the implicit scheme the formulas are:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \mathbf{v}(t + \Delta t)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t F(\mathbf{v}(t + \Delta t), \mathbf{x}(t + \Delta t), t)$$

Here the next time step $t + \Delta t$ is implicitly given as the solution of a system of equations (hence the name). While this scheme is stable for arbitrarily large time steps, now an algebraic system of equations has to be solved each time step.

By combining parts of the backwards scheme into the forward scheme a simple improvement can be made. While this forward-backward scheme is still explicit it is more stable than the standard forward scheme, without the added computational overhead generated by the backwards scheme.

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t F(\mathbf{v}(t), \mathbf{x}(t), t)$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \mathbf{v}(t + \Delta t)$$

3 Previous work on procedural destruction

Besides performing a survey of literature, a questionnaire was sent to game developers regarded destruction in games. From this questionnaire a set of criteria where determined, which according to these game developers are the most important aspects of a procedural destruction approach. The five most important aspects were control, speed, ease of use, ease of integration, and the quality of the result. One aspect that was not deemed important was realism, as long as the result looks good and convincing it does not matter if the way in which that result was achieved has any basis in reality.

From the survey of existing literature, most methods found are one of two types, namely finite element or shape matching, but a few papers found presented completely new approaches. In this chapter a selection of papers is discussed. Section 3.1 discusses some papers based on the finite element approach. Section 3.2 discusses some papers based on shape matching, and in section 3.3 some alternative approaches are presented, which do not fall into the previous two categories. For each method it is explained how they work, what the advantages and disadvantages of the method are, and how well they fulfil the criteria. Finally in section 3.4 the suitability for games of all presented approaches is compared.

3.1 Finite Element

Finite element methods (FEM) have been used in computer graphics and mechanical engineering to analyze material properties of objects for many years. FEM use meshed models to approximate how forces propagate through a continuous volume, usually the inside of an object. In (3), extensions to existing techniques are proposed for simulating crack initiation and propagation in the three-dimensional volume of objects. They relate external forces working on an object, to internal forces within the object, using the properties of the objects material. By analyzing these internal forces, the simulation can determine where and in what direction cracks are initiated, and then propagate them.

Once the internal forces have been calculated for a certain node in the FEM mesh, they decompose the forces on this node into a tensile and a compressive component. These

tensile and compressive components are then combined into a mathematical representation, representing a separating force, discarding any unbalanced portions. This separating force will then indicate whether or not material failure has occurred at this node, and from this force the fracture plane can then be determined. Finally the node is split along this plane, and the algorithm continues by calculating the forces on the remaining parts of the mesh.

This method generates very realistically looking results, see Figure 3.1, because of the use of this fracture plane when splitting nodes, based on the applied forces. Even though it is much less physically accurate than used in mechanical engineering, it is still very slow, typically taking hours of simulation time to calculate a single second of the simulation. Average simulation time, per second of simulation, for the animation seen in Figure 3.1 was 347 minutes. Also, the splitting of nodes results in a substantial increase in the number of nodes in the simulation as fractures occur, resulting in a further increase in needed simulation time. The material properties are defined by very abstract parameters, while most have a relation with actual physical properties of a real material, they are not intuitive to use, nor give a clear indication of what the result would be of changing the parameter.

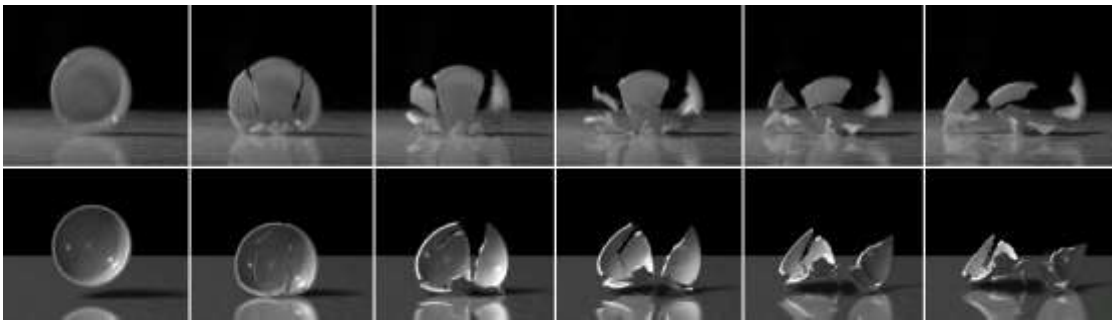


Figure 3.1 Comparison of a real-world event, top, and simulation, bottom. Image from **(3)**.

Another downside to this algorithm is that its exact workings are fairly complex. The forces being applied to an object are represented by two second order tensors. One represents the strain, the force being applied, and the other represents the strain rate, the rate at which the strain is changing. In the same way, the internal forces are represented by another set of tensors which represent the resultant forces, called stress. The elastic stress is the result of the strain, the viscous stress is the result of the strain rate. These two sets of tensors are related to each other through the properties of the material, represented by two fourth order tensors, one for the strain/elastic stress and one for the strain rate/viscous stress.

As a force in a 3-dimensional world is represented as a vector, all tensors used to describe a force become 3x3 matrices. This results in the tensors describing material properties containing 3x3x3x3 = 81 coefficients each. However, because all tensors are symmetrical, this number can be reduced to 36, and by further constraining the material to be isotropic it can be even further reduced to two independent coefficients. As described earlier, once internal forces have been calculated, in this case the internal stresses, they are decomposed and combined into a single tensor, the separation tensor. If this last tensor contains a positive eigenvalue that is larger than the material toughness, material failure occurs at the current node. The fracture plane is then generated perpendicular to the eigenvector corresponding to this positive eigenvalue, and the node splits along this plane.

A tensor is a generalization of the concept of vectors and matrices. A tensors order is the number of indices; therefore a scalar is a tensor of order 0, a vector is a tensor of order 1 and a matrix is a tensor of order 2. The rank of a tensor is the same as rank of a matrix, i.e., the maximum number of independent columns/rows.

Tensors are also used to specify how to relate the values in one tensor to another, for instance, to relate two second order tensors, a and c, to each other one fourth order tensor, b, is required:

$$a_{ij} = \sum_{k=1}^{3}\sum_{l=1}^{3} b_{ijkl}c_{kl}$$

Criteria

Control	✓✓	Material properties are stored per node, therefore local changes can be made to the material, influencing the fracture pattern by for instance weakening certain areas.
Speed	!!	Calculations take several hours to complete, even for simple objects.
Ease of use	!!	Meshes are processed automatically, but the simulation is controlled through abstract material settings, which do not have a clear effect on the resulting animation.
Ease of integration	●	Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation	✓✓	Resulting animation looks very realistic.

FEM-cube

Traditional FEM uses tetrahedral mesh elements, as those enable the arbitrary approximation of all triangular polygonal surface meshes. In (4) a FEM is proposed which uses a cube shaped mesh element, and is based on the previously discussed method with the purpose of modifying it for use in games. Cube meshes can be generated quickly as they are very simple shapes, and they have a lower memory requirement than meshes composed of more intricate shapes like tetrahedrons. Each cube has the same size and three integer position indices with respect to the bounding box of the surface, e.g., the cube in the lower

left front corner has index (0,0,0), the one left of that (1,0,0), the one above it (0,1,0) and the one behind it (0,0,1), etc, assuming (x,y,z) as indexing. It also contains a list of 8 pointers to its vertices, which are shared between adjacent cubes, and a list of pointers to triangles of the surface which intersect the cube. Each vertex has three attributes, the position, velocity and mass. The cubes are all stored in a hash table so cubes can be retrieved based on the position indices in $O(1)$ time, and as cubes store references to the triangles they intersect, spatial queries can be answered in $O(1)$ time too.

An added advantage of using cubic elements is that all elements have the same geometry, and therefore their stiffness matrices only depend on the properties of the material. This results in only needing a single matrix per material, instead of per element. As in (3), the authors use the internal stress tensors of each cube to fracture the mesh, however instead of splitting nodes arbitrarily, they separate the mesh along the cube boundaries, depending on which side of the fracture plane the element is located. Normally, FEM-based techniques are not very well suited for simulating rigid materials, as the required equations can only be solved using very small, and therefore very many, time steps. To solve this problem they simply animate meshes using rigid body dynamics, until an external force is applied exceeding a certain threshold, at which point the deformations are computed and the mesh fractured, if necessary. During fracturing, no surface triangles get split, so the process of closing the surfaces is not as straightforward as usual, but provided all intersected triangles are know through the mesh elements and these mesh elements are all cubes, it is still a relatively simple procedure. To prevent artefacts from surface triangles that are much larger than the mesh elements, surface meshes are pre-processed and large triangles subdivided along their longest edge until no triangles exist with edges longer than the cube size. The advantage of this method is that during runtime there is no need to interpolate positions and texture coordinates of existing surface elements, and the amount of new triangles generated is limited. However the resulting fractures are very dependent on the triangulation of the source mesh, and due to the underlying cube structure, fracture patterns have the tendency to look very angular, see Figure 3.2.

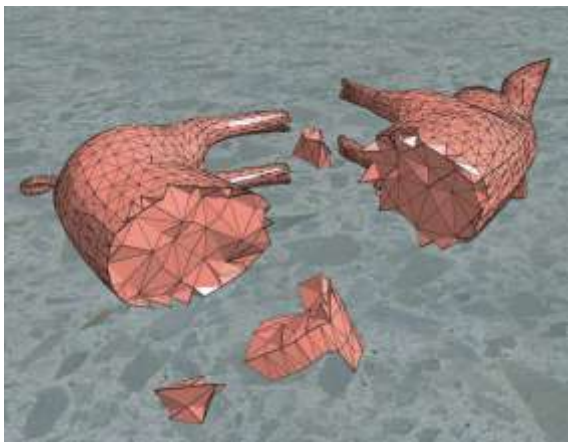


Figure 3.2 Pig model, fractured as it hits the ground. Image from **(4)**.

Criteria		
Control	●	Due to the usage of a single material matrix, any local control over the mesh settings is lost. Cube sizes are uniform and therefore cannot be changed locally either, to accommodate differences in level of detail across a mesh.
Speed	●	Real-time results for fractures, elastic deformations are also possible but these are fairly expensive and impact performance dramatically.
Ease of use	!	Meshes are processed automatically, but the simulation is controlled through abstract material settings, which do not have a clear effect on the resulting animation.
Ease of integration	●	Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation	●	Resulting animation shows several artefacts due to the underlying regular simulation mesh.

FEM-GPU

In (5) a FEM is presented, which runs on the GPU. They start by defining their objects as having a rigid inner core, and an outer elastic surface layer. The outer layer is mapped onto a texture atlas (a single large texture formed by combining smaller textures into a single texture space), encoding the shape of the objects into a texture. Each texel in this map represents a surface vertex, but actually implicitly maps to two points, the one on the surface of the object, and the other on the surface of the inner core. Due to the regular sampling, this map implicitly describes a tessellation of tetrahedral elements (see Figure 3.3), which can be used in a FEM solver.

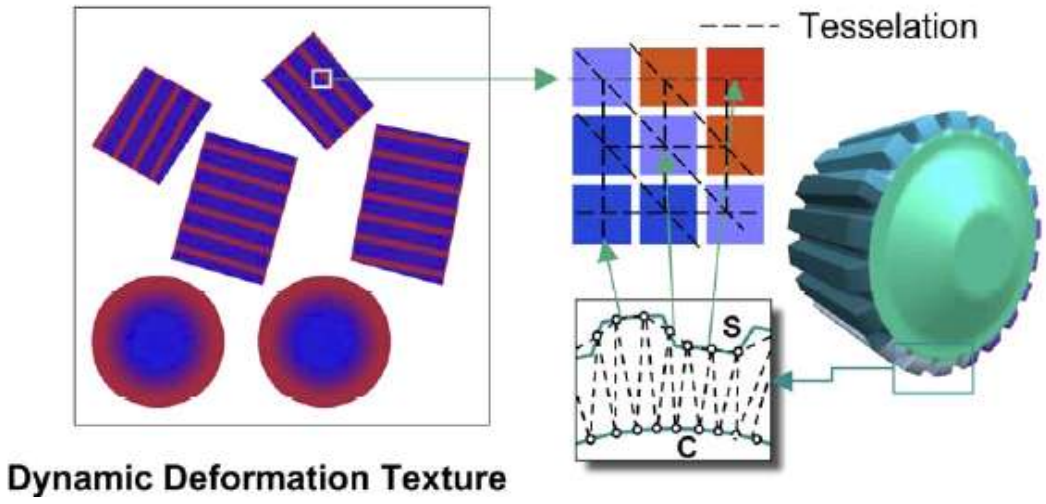


Figure 3.3 Dynamic deformation texture representation and implicit tessellation. Image from **(30)**.

By using highly-parallelizable implementations of solvers for FEMs, the can exploit the enormous processing power of modern GPU's to relatively quickly evaluate the huge number of equations. Because everything is encoded in texture atlases, GPU – CPU communications are kept to a minimum, avoiding a possible bottleneck in the system.

Besides solving the FEM parts on the GPU, the GPU is also used to find collisions and determine the proper collision responses. First a contact plane between two colliding objects is found, followed by the rendering of the collision information into this plane. The resulting image can then be transformed to the deformation texture and used for further processing.

The achieved results look very good, and allow for rich deformations of high detailed objects. However the method is limited to deformations and results are only shown for highly detailed objects. But the biggest contribution of this paper is the GPU based solution to the FEM approach.

By exploiting the ability of GPU's to execute large amounts of parallel computations the presented method is three to ten times faster than comparable methods.

Criteria		
Control	!	Material properties can be locally specified.
Speed	!	While faster than comparable methods, results are not real time.
Ease of use	! !	Meshes are processed automatically, but the simulation is controlled through abstract material settings, which do not have a clear effect on the resulting animation.
Ease of integration	●	This method requires extensive knowledge of shaders and the related programming languages. The method also requires models to be triangle stripped in a certain way which might make it less suited for use with arbitrary meshes.
Quality of animation	✓ ✓	Resulting animations are very detailed, but results are unknown for models with smaller polygon counts.

3.2 Shape matching

Where FEM's use meshed models and are based on the idea of simulating the physical properties of a material, shape matching methods are geometrically based and use meshfree models. As a result, shape matching methods do not have to store extensive additional data structures, or perform a lot of expensive decompositions and computations as they work directly with the shape of objects.

In (6) a shape matching method is proposed for meshless deformations. Their approach differs from FEM in that they replace internal energies by geometric constraints, and forces by distances of current positions to goal positions. The basic idea of their algorithm is very simple. The particles of the simulation mesh are moved under external forces, and then the original configuration of all mesh points is matched to the new positions to determine the optimal translation and rotation, which determine the goal positions of all particles (see Figure 3.4).

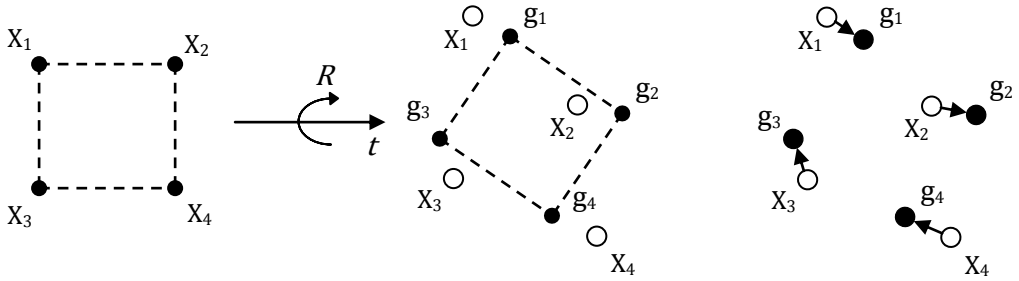


Figure 3.4 The original shape X is matched to the deformed state with an optimal translation *t* and rotation *R*. Deformed points are then pulled towards the matched shape g.

With these goal positions known, all particles are moved towards their goal position. For this they use a modified explicit forward-backward Euler integration which results in a unconditionally stable system, as the simulation points are always moved either in the direction of, or directly to, their goal position, depending on a stiffness factor α (see Eq. (1)).

$$\mathbf{v_i}(t + \Delta t) = \mathbf{v_i}(t) + \alpha \frac{g_i(t) - x_i(t)}{\Delta t} \quad (1)$$

With α = 1 the simulator imitates rigid bodies and only a few vertices would need to be animated as particles, as the remaining vertices can be transformed using the computed transformations. Contrary to the previously described finite element methods, this method is well suited for stiff or nearly rigid objects. However in its basic form, it is less suited for objects undergoing large deformations. To extend the range of motion, linear and quadratic deformations, like shearing and twisting, where implemented. This was achieved by instead of only calculating the optimal rotation and translation, calculating the optimal transformation. The range of motion was extended even further by creating disjoint clusters of points, which are all individually matched to their original shape, and add a term $\Delta\mathbf{v_i}$ to each of the particles they contain.

While the presented implementation is fairly fast and able to simulate dynamics for a couple of hundred objects in real-time using the linear and quadratic deformations, cluster-based shape matching results in very visible artefacts in the animation of objects (see Figure 3.5, left and middle), especially under extreme deformations. Also, the more clusters are used the higher the computational complexity of the simulation becomes, as well as the computation time, quickly reducing the advantage in computational complexity.

Criteria		
Control	●	Entire simulation is globally controlled through a few parameters. No local changes can be made to material properties.
Speed	✓	Allows for the simulation of simple deformations of several hundred simple objects in real-time, but speeds become considerable slower when using the more complex deformations.
Ease of use	✓	Meshes are processed automatically. The parameters have an intuitive effect on the simulation.
Ease of integration	●	Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation	✓	Unconditionally stable even under extreme deformation, however extreme deformations show very visible artefacts when using clusters.

Lattice Shape Matching

In (7) a Lattice Shape Matching method is proposed. The basics are the same as in (6) as it is based on the research described in that paper. The presented approach extends the system of using clusters to shape match subsets of points of the model by instead of using a relatively low number of disjoint regions, forming a lattice of multiple overlapping regions. Each particle in the system stores a one-ring neighbourhood of particles which share at least one lattice cell. Each region determines its own optimal rigid translation and rotation, and particle movements are weighted depending on the number of regions they belong too, resulting in smooth transitions between the different regions, see Figure 3.5. Rigidity of the object can be influenced by the size of the regions, in theory creating a fully rigid object if the region size is set to the size of the object.

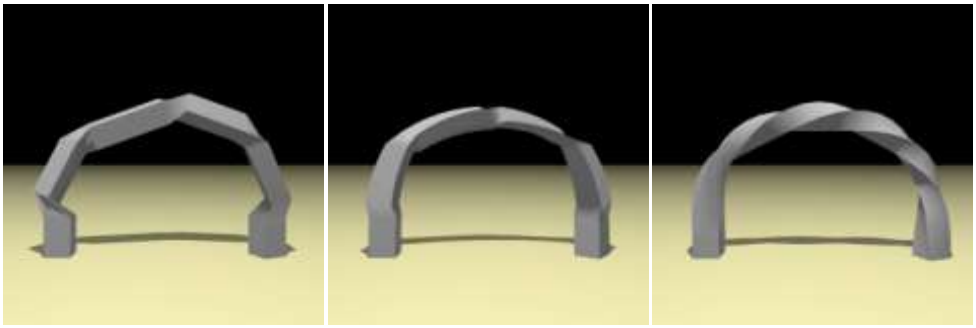


Figure 3.5 Comparison of shape matching methods under extreme deformation:(Left) Linear and (Middle) quadratic shape matching with a low number of regions; (Right) lattice shape matching. Image from **(7)**.

Each region sums over its contained particles, as it is looking for its optimal translation and rotation. However, as a lot of regions overlap, breaking down each regions particle list into sub-regions, every region can reuse the calculations done on these sub-regions, resulting in O(w) cost for the shape matching, instead of O(w^3), where w is the half width of a region. By

expanding the summations and observing recurrences of certain calculations in these summations, even more calculations can be reused, reducing the total amount of summations down to 6, independently of the region size, resulting in O(1) calculations.

The optimal translation for each region is found by looking at the centre of masses, which can be computed when the regions are created. To find the optimal rotation efficiently, a fast method to compute polar decompositions was used. Like (6), cyclic Jacobi iterations where used to find the eigenvalues and eigenvectors of the rotation matrix. By initializing the search matrix with the resulting eigenvector from the previous decomposition, instead of the identity matrix, this process was sufficiently sped up to avoid a potential bottleneck.

Criteria		
Control	●	Entire simulation is controlled through a single parameter. In case of fracture, a 2nd parameter is needed, however both parameters only allow for global influence of the behaviour of the simulation.
Speed	✓✓	A few hundred objects can be simulated in real time.
Ease of use	✓	Meshes are processed automatically. The parameters have an intuitive effect on the simulation.
Ease of integration	●	Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation	✓✓	Very good, extreme deformations do not result in artefacts, however due to regular distribution of simulation particles, areas of higher level of detail cannot be simulated properly, without increasing particle count in other areas where it is not needed.

3.3 Alternative approaches

Graph-based

The methods previously described in sections 3.1 and 3.2 are quite complex and tend to be controlled through abstract parameters, without intuitive ways of influencing the way an object fractures. In (8) an original method is presented for procedural creation of cracks and breaking objects into fragments. In this method a designer specifies a two dimensional crack pattern, which gets mapped onto the surface of an object to create a geometric skeleton. By sweeping a profile curve along this skeleton, a carving volume is created which allows cracks to be created using Boolean difference operations on the original model and the carving volume. Different profiles for the sweep curve result in different visualizations of the cracks. To fracture an object the user specifies a fracture pattern, which specifies the fracture between two fragments. The shape of the fragments can be controlled through a set of simple parameters. The first parameter α, specifies the angle between the principal axis of the object and that of the fracture pattern, therefore a small α will result in thinner longer

fragments than a large α (see Figure 3.6). The second parameter ΔV, specifies the volume ratio between two fragments; this influences the distribution of the size of the fragments. In Figure 3.6 this would be shown by moving the fracture patterns up or down the principal axis. While the presented figures show results which look convincing (see Figure 3.7), sadly no interactive demo or moving images are available, so it is hard to judge how the final results would look when they are animated. Also while their method is fast compared to traditional methods, even for a relatively low number of fractures the method takes a few seconds to generate a result.

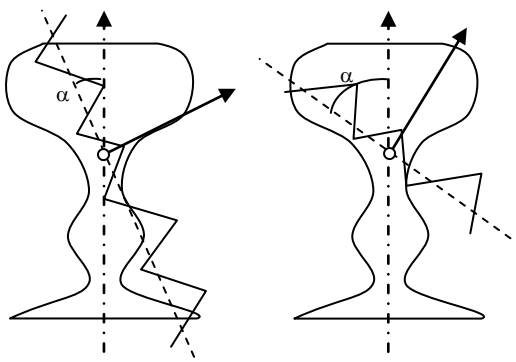


Figure 3.6 The shape of fragments can be controlled by selecting the orientation of the fracture pattern relatively to the principal axis of the object.

Figure 3.7 A scotch glass, a statue and a flute glass broken into 18, 128 and 48 fragments respectively. Image from **(8)**.

Criteria

Control	✓✓	Fracture patterns defined by a graph created in a pre-processing stage. Application of this pattern can be influenced through two parameters.
Speed	!	Simulation requires several seconds to process an object.
Ease of use	✓✓	Creating a fracture pattern is a very intuitive way of defining how an object breaks. Additional parameters have a clear purpose and effect on the simulation.
Ease of integration	●	Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation	✓	Provided still images look good.

In (9) a different graph based approach is presented. They use a spider web like crack pattern, a central point with interconnected lines radiating outwards, to break objects. The nodes in the pattern are randomly translated based on material settings, as are the number of lines in the pattern. The pattern is then extruded perpendicular to the plane in which is it defined in order to create which is then used to cut the target object. This approach creates

good results when cracking flat objects like a plate of glass, but due to the straight extrusion of the crack pattern fragments created of three dimension objects do not appear realistic.

In (10) an approach is presented based on (9). As Fox a spider web crack pattern was used, but instead of only using a single pattern, several patterns are placed on top of each other, connecting them together to form a honeycomb like structure which is then used to cut a target object into blocks. A Binary Space Partitioning was implemented to quickly solve spatial queries on triangles in order to decrease the computational complexity of the prototype. While the visual results did improve over those of Fox, due to this approach no longer generating elongated fragments, the performance did not significantly improve nor did the approach offer any more (or less) control over the simulation or influence over the design of the crack pattern.

In (11) an approach is presented which is based on both (9) and (10). As Miller, Workman created a three dimensional crack pattern, but instead of using vectors to represent the connections between nodes, a grid like approach was used to generate a crack pattern. The crack patterns were generated from a given alpha texture, giving a designer a larger amount of control over the design of the crack. However, the crack patterns that were generated were not very well suited for most cracks, due to the grid based approach that was taken. Sadly they did not actually succeed in generating a usable crack model with this approach; instead a handmade crack pattern was used to test the fracturing algorithms. The visual results were similar to those achieved by Miller, due to the similarities of their approaches; however the performance of Workman's approach was worse than that of Miller but this is likely due to the specific implementation and not the approach itself. While the approach did offer more control to a designer as compared to the other approaches, the grid based generation of a crack pattern did not result in crack patterns of sufficient quality.

Spring constraints

In (12) a technique is proposed which is a combination of FEM and a mass spring system. As in a FEM a tetrahedral mesh is still generated, but instead of performing the expensive computations of a FEM, a rigid mass spring system is approximated using distance constraints. By doing this, the problem of simulating very stiff springs is circumvented, which can normally only be done in very small time steps, and what would therefore be computationally very expensive.

These distance constraints are specified between the centres of mass of neighbouring tetrahedral mesh elements, which is precisely what a completely rigid spring would be. Once

a distance constraint becomes violated the link is simply removed from the system, enabling objects to fall apart. To avoid unnecessary calculations the simulation is only ran during an impact, either instantly breaking an object or not. As this could result in numerous simultaneous removing of constraints, an object could be pulverized in a single frame. To enable the system to also create large fracture elements the forces exerted on the object are slowly ramped up over several smaller simulation steps, allowing weak constraints, i.e. constraints which allow for only a small distance change, to break first and allowing for the propagation of a crack over the surface. Their method almost ran in real time, but for larger amounts of constraints the simulation already took several seconds to complete. However, an advantage of this system is that crack patterns can be easily influenced by either weakening or strengthening the distance constraints between individual nodes.

Criteria		
Control		Individual constraints can be easily changed to influence fracture patterns
Speed		Faster than comparable FEM, but not fast enough for real-time applications when using a reasonable amount of constraints.
Ease of use		Individual distance constraints between simulation nodes may not have an intuitive effect on the simulation, but their use is clear.
Ease of integration		Simulation only requires a surface mesh as input, and will generate surface meshes as output.
Quality of animation		Quality of the animation is largely dependent on the triangulation of the simulation mesh as mesh elements are simply broken off to be animated separately and no mesh elements are split.

3.4 Suitability for games

With all methods reviewed based on the same set of criteria, they can be compared on how well they perform regarding each criteria, see Table 1. While the finite element methods generally offer a large amount of control over the simulation and generate a very high quality animation, they are also generally the slowest methods and use abstract non-intuitive material parameters to control the behaviour of their models. The shape matching methods, on the other hand, require a lot less computational power, while still maintaining a high quality of animation. While they offer less control over the simulation, they are controlled through a few simple parameters that have a clear influence on the behaviour of the model. The alternative approaches are usually somewhere in the middle, they are slower than the shape matching methods, but they are generally faster than finite element methods. The graph-based method in particular offers a large amount of artistic control and is easy to use with a few, but intuitive, parameters.

	FEM	FEM-cube	FEM-GPU	Shape matching	Lattice shape matching	Graph-based	Constraints
Control	✓ ✓	●	!	●	●	✓ ✓	✓
Speed	! !	●	!	✓	✓ ✓	!	!
Ease of Use	! !	!	! !	✓	✓	✓ ✓	!
Integration	●	●	●	●	●	●	●
Quality of animation	✓ ✓	●	✓ ✓	✓	✓ ✓	✓	●

Table 1 Summary of scores of methods, per criteria, relative to each other.

All methods score the same on integration, as all methods take a surface mesh as input and generate more surface meshes as output. The integration into an existing pipeline of any of these methods will therefore be equally difficult, but should not pose any major issues as objects in games are usually represented with a surface mesh.

4 Procedural destruction in gaming

In the gaming industry there have also been developments with regards to procedural destruction. In the past few years several middleware companies have released products for procedural destruction. Currently there are three major middleware companies that develop physical simulation software, which includes basic physics like rigid body dynamics and ragdoll animation, but also encompasses more complex techniques like cloth dynamics and procedural destruction.

4.1 Havok

Havok (13), launched in 2000, is one of the older middleware companies specializing in physics simulation for games. It originally started as a basic physics simulator featuring rigid body dynamics. In 2003 ragdoll animation was added to their simulator, which at that time was a very new and cutting edge technology. Havok's product line has since then expanded to include animation tools, clothing dynamics (2008) and also destruction tools.

In the Havok Destruction product brief (14), several tools are described for destroying objects, buildings, structures like bridges or scaffolding and object deformation. Sadly the product brief does not provide any useful insights into how Havok's technology works, but on the product showcase website (15) several videos are available that showcase the results of these tools.

As of May 2008, Havok's basics software package is freely downloadable. While this does not include the Havok Destruction package, it does contain several demos showcasing how one can implement destructible object using Havok. One of these demos is of a destructible brick wall. The wall starts out as 3 simple stretched boxes, but as it gets hit with a block shot by the user, bricks start coming loose and fall off. As soon as enough force is applied to separate a single brick, the wall is cut twice horizontally into 3 sections, where the middle section is the row containing the brick to be separated, as seen in Figure 4.1. After this, the row of bricks is cut twice vertically to create the single brick that needs to be removed. The single brick is then inserted into the rigid body simulation and animated as a separate object. The remaining objects, which together form the remainder of the original wall, are 'glued'

together using constraints. As more blocks are shot against the wall, this process is either repeated to separate more bricks, or some of the constraints keeping the separate wall pieces together are broken, making it possible to break off entire sections of the wall.

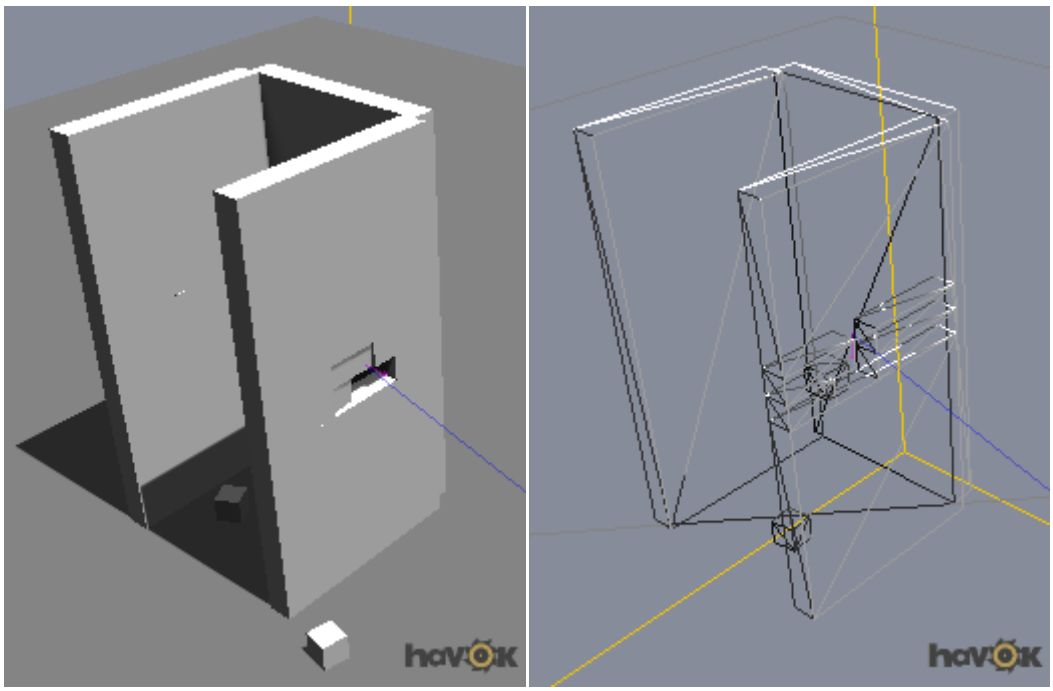


Figure 4.1 left: Visible surface mesh, right: underlying simulation mesh, showing subdivided surfaces as bricks are broken off.

This technique of using simple planes to cut the original object can be used to create a large variety of fracture patterns (see Figure 4.2), however all this has to be coded by a programmer. Another downside to this technique is that the patterns are easily recognizable, although by adding some sort of randomization this problem should not be that hard to work around. Presumably, the aforementioned fracture generator included in the Havok Destruction encapsulates this functionality into a more user friendly tool.

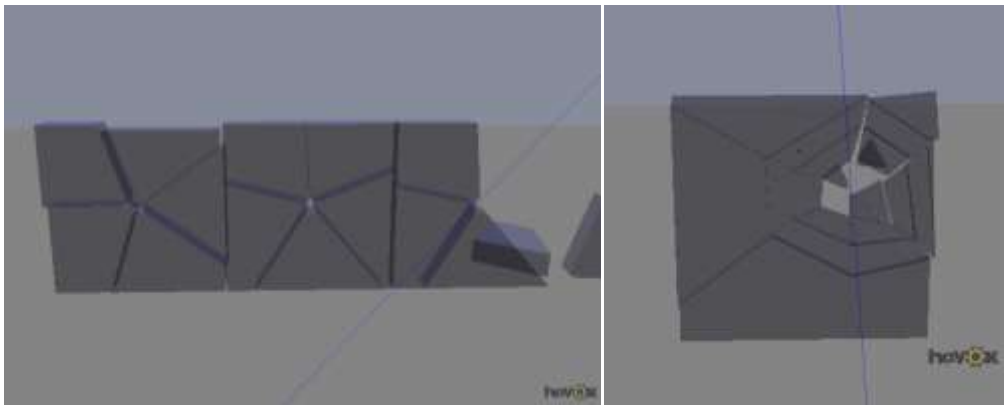


Figure 4.2 Different fracture patterns can be created.

4.2 Digital Molecular Matter

The latest competitor in the field of physics simulation middleware is Pixelux (16); in 2008 they unveiled their proprietary destruction technology called Digital Molecular Matter (DMM). DMM uses a tetrahedral mesh to represent objects and applies the Finite Element Method to this mesh to simulate material physics in a more realistic fashion than was possible before. Their implementation is based on the thesis work, and build with the help of, James F. O'Brien. What specific changes and optimizations they have made to the algorithms described by O'Brien is not known, but they are able to simulate several thousand of nodes in real-time, judging from their Maya plug-in. However the computational complexity is still fairly high, so using it on too many objects at the same time results in unsuitable frame rates for real-time applications.

DMM was first featured in a commercial product in mid September 2008, namely in LucasArts' Star Wars: The Force Unleashed. When playing the demo, the limited use of DMM becomes quickly apparent. It was limited to a few glass windows in a corridor that could be shattered, and the odd metal door blocking a corridor that had to be bent, in order to be able to pass through. However, slicing a droid in half, or ripping piping off the wall is still animated using traditional scripting and art swap methods, which can be clearly seen as each time you slice a droid, the result looks exactly the same.

Figure 4.3 A DMM enabled castle is bombarded with rocks and crumbles as it is hit.

4.3 PhysX

In 2005 Ageia, now part of Nvidia, released their PhysX (17) SDK, a software platform for physics simulation. It included advanced technology like cloth and soft body dynamics, which were new to gaming technology at that time. For their soft body dynamics they use

tetrahedral meshes as a volume representation for the simulation, generated from the surface mesh of an object (see Figure 4.4). By performing a simulation on the underlying tetrahedral mesh, the visible object can be animated. Object rigidity is controlled primary through two constraints, a volume constraint and a stretching constraint. Object tearing is present in the current version as an experimental feature. The tearing is controlled through a tear factor, once edges become longer then the tear factor times their base length they are separated and new elements are created. By setting the volume and stretching constraints to their maximum value, allowing no changes to the volume and dimensions, the soft body simulation can be turned into a rigid body simulation, and the tearing could be used for destroying rigid objects as well. However, this tearing functionality was listed as being an experimental feature, and at the time appeared to be non-functional. Another note is that while the simulation runs at real-time speeds for single objects, composed of tens or up to a few thousand mesh elements, the simulation slows down considerably when multiple objects are added.

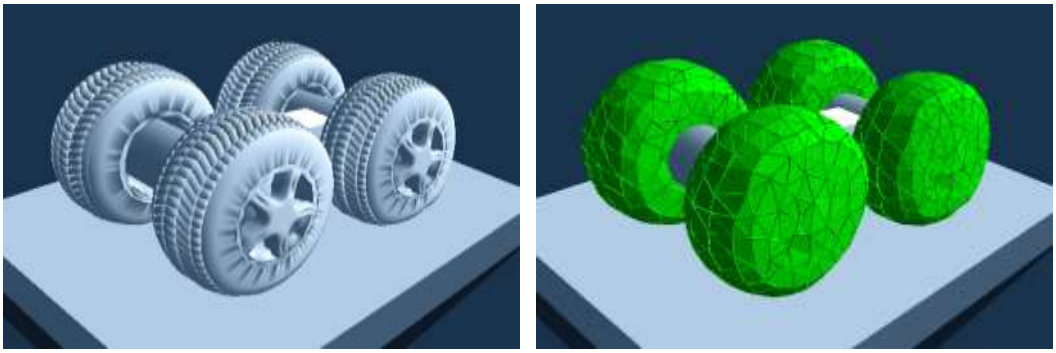


Figure 4.4 Surface mesh and the underlying volumetric simulation mesh.

Once the total number of simulation elements in the scene approaches five thousand elements, the frame rate already drops below 60 on a high-end PC, and this is in a bare environment with simple shading and no complex game mechanics running in the background.

With the introduction of Nvidias CUDA technology, the general purpose processing units of their video cards can be used to perform various tasks other then pure rendering, like physics calculations. Due to the massive increase in processing power of modern video cards and the ability to quickly perform large amount of parallel computations technologies like PhysX and DMM are quickly becoming the go-to standard of the games industry. However, while CUDA reduces the problem of the computational complexity of these approaches, the lack of control and ease of use still makes these approaches less desirable.

4.4 Unreal

Unreal (18) uses PhysX as its main physics simulator. However, on top of this they have a custom fracture tool, which can be used to create destructible static objects in a scene. According to the Unreal Developer Network documentation (19), the Unreal editor contains a tool to fracture static meshes. The user can select the number of fragments to break the object into, and scale the planes along which the fragments are created. The tool then pre-computes the fragments, which are then used at runtime to replace the object once it gets damaged. However this tool offers little benefits over the traditional method, as it simply moves the process of creating the fragments from an artist application of choice, into the Unreal editor.

Sadly I have not been able to actually test the tool. While the tool has been present in the editor since the December 2007 build of the editor, it is currently only available to licensed developers, and is not included in the editor shipped with Unreal Tournament 3.

The recently released AAA title Gears of War 2, by Epic Games, was built on the latest version of the Unreal 3 engine. As Gears of War is a third person shooter game, it features a lot of explosions and things breaking. In the original Gears of War there were only a few destructible items in the game, like some crates and sofa's, which were made using the art-swap method, something which was easily spotted. In the sequel a lot more things can be destroyed or damaged, for instance almost anything the player can take cover behind, e.g. concrete blocks or tiled walls. While this cover cannot be completely destroyed, it can be damaged. When shot at, or when grenades explode in close proximity, pieces of the surface of the object are broken off, exposing the underlying internal material of the object, usually a basic type of concrete. While the results might not appear completely realistic when you take the time to analyze the animation, during normal gameplay the intensity of the game does not allow the player to worry about this, and the effect of semi-destructible objects greatly adds to the sense of being in a 'real' world, instead of in some static indestructible one.

4.5 Red faction

One of the first 3D games ever to feature dynamically non-scripted destructible terrain was Volitions Red faction (released May 2001). Their original engine used relatively simple Boolean operations, based on (20), for modifying terrain, walls and certain objects, see Figure 4.5.

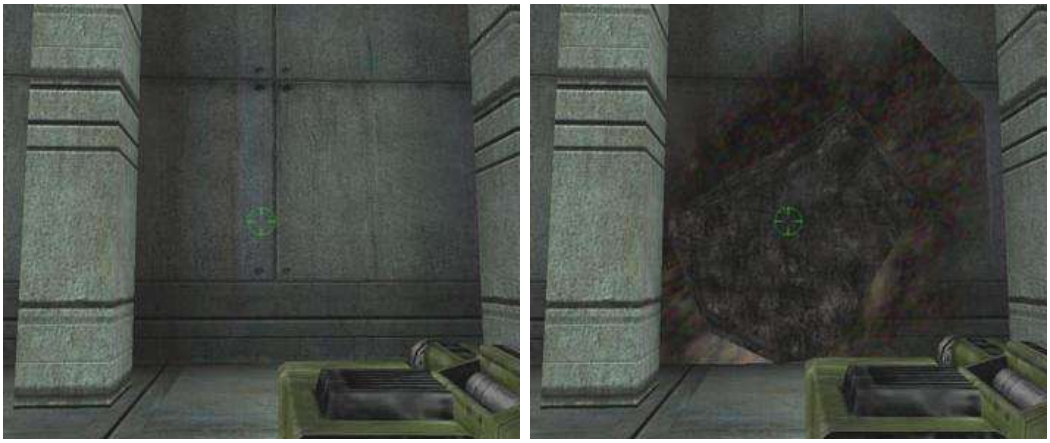


Figure 4.5 From the original Red Faction. Left: Scene before applying a geomod. Right: scene after geomodding has been completed.

The newest game in the Red Faction franchise, Red Faction Guerrilla (June 2009), features fully destructible objects and environments through a combination of proprietary destruction and stress propagation systems and the Havok physics engine. This combination of different approaches to destruction can easily be seen in the demo. For instance some objects simply break apart when destroyed, releasing the constraints that held the object together, but glass shatters into a lot of tiny pieces. Walls on the other hand break into a multitude of fragments, depending on the type of the material the wall is made off.

The stress propagation system can be seen at work when destroying buildings. This system takes material properties into account, like weight and strength, to determine if a structure should collapse or not. When the structural support of an object is removed, like the first floor of a building or the pillars of a bridge, the remaining structure comes crashing down.

4.6 Frostbite

The Frostbite Engine is the game engine used in the Battlefield series by DICE (21). The release of Battlefield: Bad Company (June 2008), marked the first commercial use of Frostbite Engine. In this version of the engine, buildings could be procedurally destroyed, however some limits where imposed on the destruction, a skeleton structure would always remain, see Figure 4.6. In the recently released sequel Battlefield: Bad Company 2 (March 2010), the destructibility of buildings has been extended allowing users to fully destroy a building, reducing it to a large pile of rubble, see Figure 4.7.

Figure 4.6 Walls of buildings could be destroyed in Battlefield: Bad Company, however a skeleton structure would always remain. Images taken from **(22)**.

Figure 4.7 Complete destruction of a building in Battlefield Bad Company 2. Images taken from **(23)**.

5 Approach description

As seen in chapter 3, for our approach to be successful it needs to provide the user with an intuitive way to create destructible objects, allow them to quickly create the desired effects, but also offer a large amount of control over the resulting animation. It will also need to run in real-time to be applicable to games and to provide interactive editing capabilities.

In chapter 4 we could see that developers are focusing largely on FEM based technologies, but that they are not yet widely used, because they lack the required intuitive approach. In some cases, very convincing results are achieved, e.g. Red Faction Guerrilla and Bad Company 2, but it is largely unknown what the underlying technology is.

To achieve this intuitive editing we introduce the concept of destructible materials.

5.1 Destructible material

Normally a material defines how something looks, but a destructible material will determine how something breaks. We define a destructible material as a collection of several crack and/or fracture patterns. These patterns are reusable by several materials, and materials are reusable by several objects. Because all components are reusable, once a library of patterns and materials has been created, making any object destructible will be as simple as selecting and assigning the desired destructible material, significantly reducing the amount of time required for creating content.

How a destructible material will actually be used will really depend on the specifics of the project it would be used in. Selecting which pattern to apply could be based on various kinds of events for instance:

Impact type

Hitting an object with rockets will likely result in a different crack than hitting it with bullets or a hammer. It should be possible to assign different patterns to the various types of impacts, or even the specific items, causing the impact. This would give a designer the ability to still create patterns separately, while the application of these patterns is handled in real-time.

Impact force

Similar to impact type, the force of the impact could also influence the pattern selection. Hitting an object with a large explosive, or driving over it with a truck is likely to have a different effect than hitting it with your fists. The magnitude of the impact force is also something that could determine the difference between cracking and fracturing an object. If enough force is applied in one impact, it is likely an object will simply fracture into pieces, instead of becoming cracked.

(Remaining) strength of the object/material

Often in gaming environments, objects and characters have a certain amount of health, or hit points, indicating the amount of damage they can absorb before being destroyed or killed. A similar system could be tied into the destruction, where each impact with an object lowers the amount of remaining health of an object, depending on a combination of the impact type and force. Once the remaining health of the object reaches zero, or some other threshold, instead of cracking the object it could be fractured, indicating that the object has no more strength left and simply shatters into pieces.

5.2 Designing crack and fractures

Designing crack and fracture patterns and the way they will be applied will be similar to the way presented in (8), where graphs are used to represent the branching structure of a crack, combined with simple parameters that define the width and depth of a crack at the nodes in the graphs. By allowing a designer to simply draw how a crack should look and converting brush strokes into graph elements, a crack pattern can be created in a very short amount of time, see Figure 5.1.

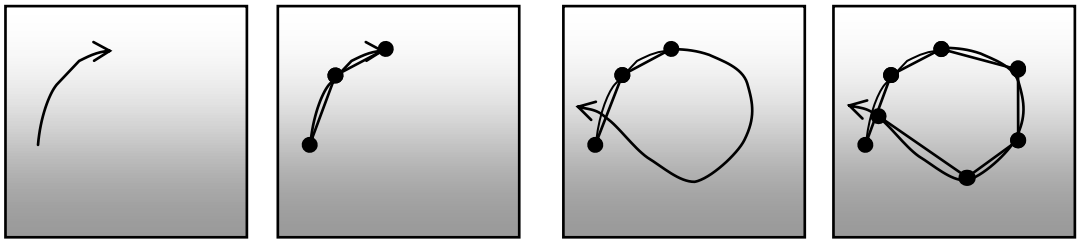


Figure 5.1 As a designer draws the strokes are analysed and turned into a graph.

To further aid a designer in achieving the result he wants, this approach will be extended to 3D. By projecting brushstrokes onto the surface of a model, they can internally be converted into a 2D format, whilst allowing a designer to create a crack that will look like he intends it when applied. As the crack pattern is still stored in 2D it can still be reused in different

materials or on other objects and further modifications can still be made by editing the 2D definition of the pattern.

When applying destruction to a target object, the patterns will be mapped onto the surface of the object and a crack volume can be generated based on the parameters of the nodes. The target object can then be cracked by performing a Boolean difference operation between the target object and the crack volume.

A fracture pattern, in turn, will define the cross section of the fault between two fragments, and a series of line segments will be used to present them, see Figure 3.6. Two simple parameters are used, the rotation and volume ratio parameters. These allow a designer to change the orientation in which the pattern is applied with regard to the principal axis, the longest dimension of the target object, as well the volume ratio between two fragments. The designer can then easily fracture objects into long thin or more square fragments with either equal size or varying from small to large. As with cracking, a mesh is generated from the pattern definition. Boolean operations are then performed between the target object and this fracture shape.

6 Prototype

In this chapter we will discuss the design of the prototype, and we will explain which functionality discussed in the previous chapter will be added to the prototype, which will not and why.

A prototype will be built in order to determine the feasibility of the approach described in the previous chapter, as well as is suitability in a real-time gaming environment.

6.1 Prototype focus

The main focus of the prototype will be the implementation of the needed algorithms to support the basic cracking and fracturing operations. These algorithms will need to be fast and efficient in order to be able to create a system that operates in real-time. Several well-known algorithms like triangle intersection and Boolean operations will need to be implemented, but customized and optimized to suit the data structures used in gaming environments. These algorithms will be combined into a library that will allow a user to easily create destructible objects in any project.

An editor will also be created which will allow for the design and preview of a destructible material. This editor will need to offer functionality for creating crack and fracture patterns in an easy and intuitive way, while using the created libraries and gaming environment to provide instant feedback to the user. The representation of a destructible material in the prototype will be simple: it will contain one list of assigned crack patterns, a list of assigned fracture patterns and a strength parameter. This strength parameter is the magnitude of the impact defining the threshold between cracking and fracturing. Any impact force with a magnitude smaller than this strength value will result in a cracking operation being performed, while impact forces with a magnitude higher than this value will result in a fracturing operation.

Once a pattern is assigned, several parameters can be set. Crack patterns are defined in a unit space to facilitate the ease of using them in different destructible materials. Once they are assigned to a material, a scaling parameter becomes available to determine the actual size of that crack pattern for that specific material. As fracture patterns are automatically

scaled to ensure they will fully intersect the target object, no scaling parameter will be available for them. However, the orientation and volume ratio parameters will be settable for each fracture pattern. When applying a cracking or fracturing operation, the actual pattern to be applied will be chosen randomly, but to provide a little more control over the pattern selection in the prototype, each assigned pattern will also have an extra parameter that can influence the likelihood of it being selected.

6.2 Development environment

The prototype will be developed using the proprietary game technology of Cannibal Game Studios. For visualization purposes, the Cannibal Engine will be used, which will allow for the rapid and easy visualization of the results of the various stages in the development process of the prototype.

The editor will be built in the Cannibal Composer framework. Cannibal Composer is a data-item centred editing environment where different types of data-items can be edited through various plug-ins. A plug-in builds on a fully customizable basis for editing/viewing/any-other-way-of-interacting-with every aspect of a data-item through simple widgets.

Three new data-items will be created: *destructible materials*, *crack* and *fracture patterns*, and a single plug-in will be added which allows for the editing of these three data-items, as described in the next section.

6.3 Editor

The prototype editor will contain all basic functionality needed to design and compose a destructible material. Essentially it will consist of three separate editors editing crack patterns, fracture patterns and destructible materials.

Editing crack patterns

Crack patterns will be created using a simple graph editor where nodes can be added to a graph by simply clicking on a canvas, see Figure 6.1. This will create a connection between a new node placed at the clicked location and a previously selected node. Nodes can be moved by click-dragging and once selected, the properties of a node, as described in section 5.2, can be edited using sliders, see Figure 6.1. To quickly provide feedback about the general shape of the resulting crack volume, these properties will be used to draw the approximate size of the volume around the graph; this will quickly alert the user to any erroneous input values.

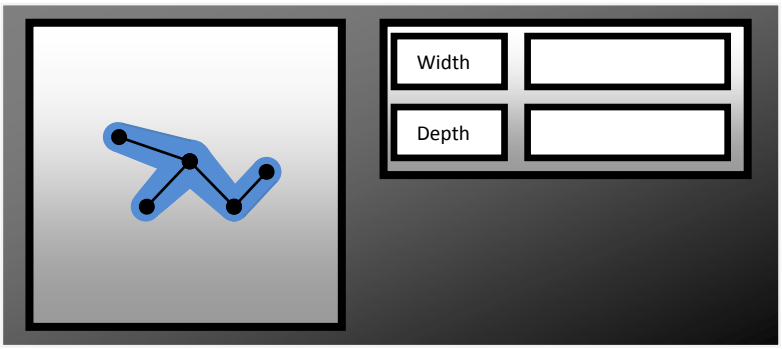


Figure 6.1 A simple graph editor consisting of a canvas on the left and options to edit node properties on the right.

Editing fracture patterns

Fracture patterns are represented by connected line segments, essentially a graph where each node has at most two connected edges, see Figure 6.2. In the prototype a fracture pattern will start as a single horizontal line segment bound by two nodes. Additional nodes can be added, and removed, and will be equally spaced along the original segment. Every individual node can be moved up or down, but their horizontal position is strictly determined by the number of nodes in the pattern. While there is no constraint imposed on

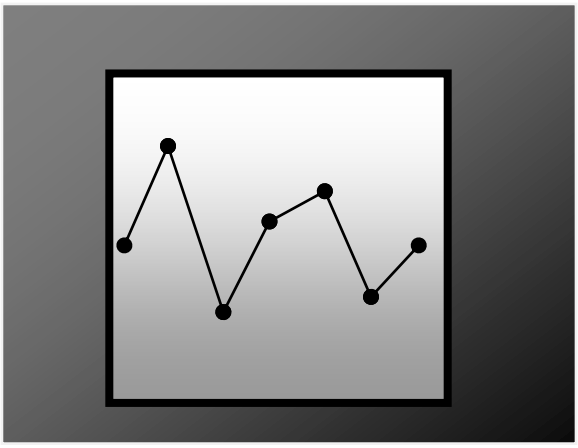


Figure 6.2 A simple fracture editor consisting of a canvas.

this editing paradigm by the algorithm, this approach simplifies the data model by only having to store the Y components of the node positions, but it also allows for the reuse of pre-existing UI elements present in the editing environment.

Editing destructible materials

The editor for the destructible materials will feature two separate modes, one for *composing* the destructible material and one for *testing* it. In compose mode a library of previously created crack and fracture patterns will be visible, see Figure 6.3, allowing a user to quickly

apply any previously created patterns to the current destructible material. A preview display will show any pattern currently selected, to aid the user in determining exactly which pattern he has selected. Any desired pattern can then be dragged onto the appropriate list from the library to assign it to the material. Once assigned, the various properties, as described in section 6.1, can be set using sliders that appear in the bottom left widget. Finally a simple slider will be available to set the material strength.

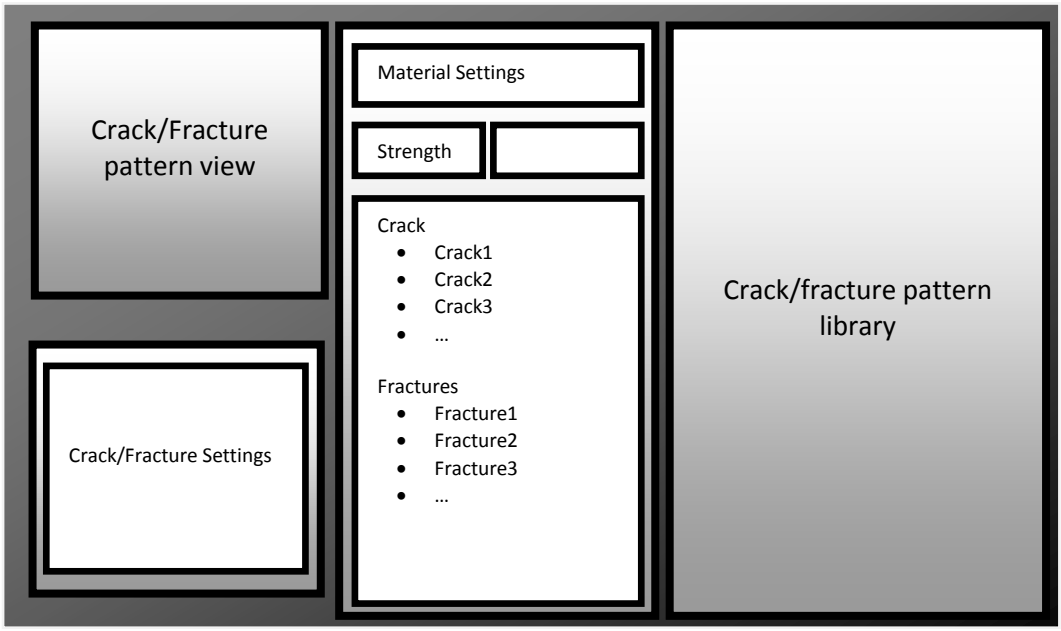


Figure 6.3 The destructible material editor in compose mode.

In the testing mode all UI elements relating to the editing of the material will automatically disappear and new UI elements consisting of a 3D viewport, various testing options and a library of loaded models, will appear, see Figure 6.4. The viewport will allow a user to interact with various objects, applying destruction to them using the created material. The magnitude of the impact force can be set using a slider. The type of object to test the material on can be selected using a radio group, selecting a standard cube, a sphere, or any other object that can be freely assigned from the content loaded into the editor. By then clicking on the viewport, a destruction event is triggered. Depending on the setting for the magnitude of the impact force the user has entered, either a cracking or fracturing operation is performed and the result is visualized. Repeated operations can be performed as desired and if the user is unhappy with the result, additional changes can be made to the material by instantly switching back to the editing mode. Lastly a reset button will be present that will reset the displayed object.

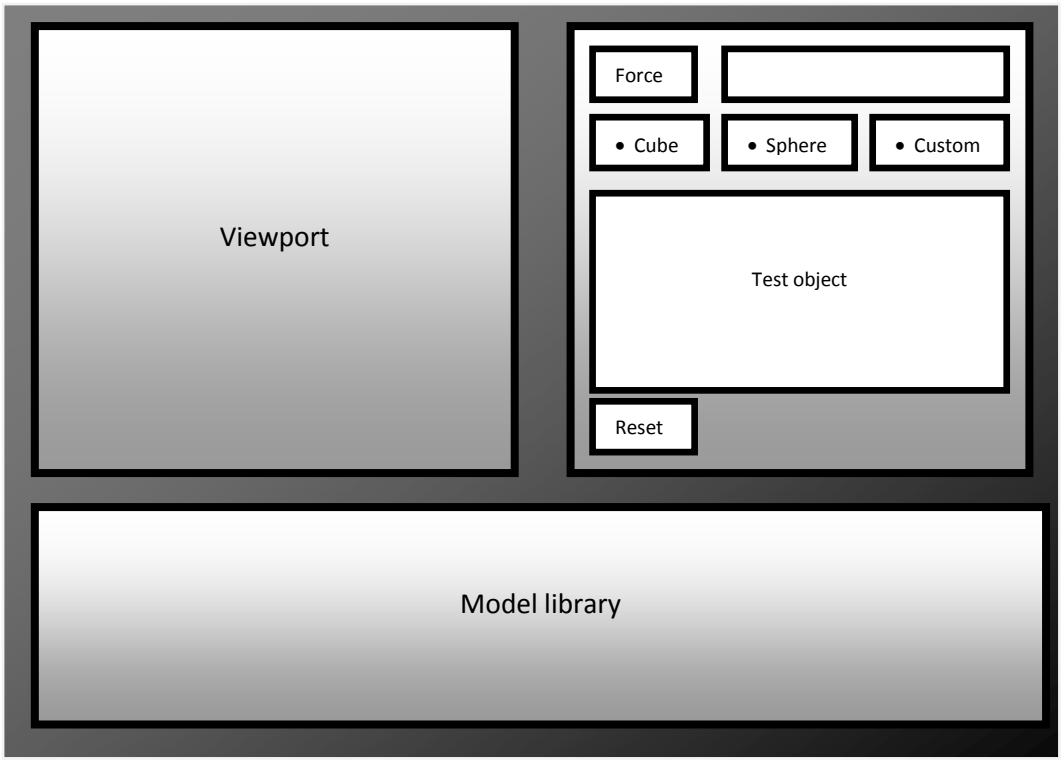


Figure 6.4 The destructible material editor in preview mode.

7 High level design

In this chapter we will discuss the architecture of the destruction algorithm. Section 7.1 will give a description of the different steps of the procedure and what their inputs and outputs are. In section 7.2, cracking will be discussed, followed by fracturing in section 7.3. Lastly section 7.4 contains a discussion on the Boolean operation.

7.1 Algorithm overview

At the highest level there are only a few steps involved in the entire procedure, see Figure 7.1. The first step is the input step in which the user performs an action, like clicking in the viewport in the editor. The target object and the impact information, which consists of:

- the force of the impact,
- the triangle which was hit by the ray impact,
- the location in that triangle where the object was hit,

are then fed to the second step which determines the type of destruction to perform and selects an appropriate pattern from the destructible material assigned to the object. This pattern and the target model are then fed to either the cracking or fracturing sub-routine which generates a destruction mesh. The models created in these steps are then used in a Boolean operation together with the target model, after which the resulting model(s) is/are returned (in the case of a fracture n fragments are created).

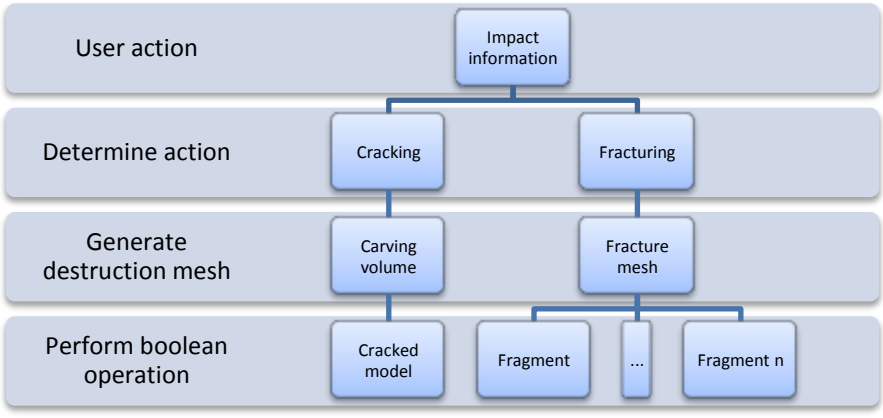


Figure 7.1 High level overview of the entire destruction procedure.

7.2 Cracking

The cracking sub-routine consists of 2 steps, see Figure 7.2, and requires as input the target model, the crack pattern to apply and the impact information as described in the previous section.

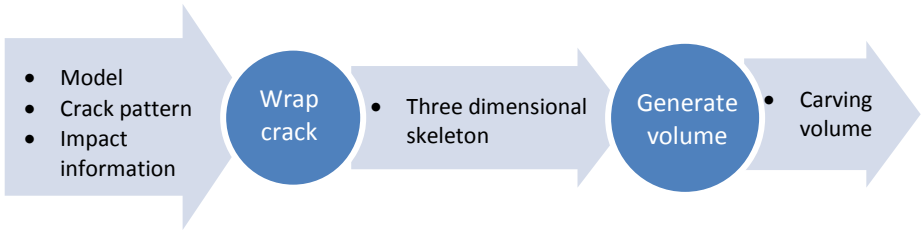


Figure 7.2 High level overview of the cracking algorithm.

The impact information is used to wrap the given pattern around the target model to create a three dimensional skeleton. The wrapping starts by placing the first node of the pattern on the surface of the given model at the impact location. One by one the edges in the pattern are projected onto the surface of the model. Projecting these edges consists of a loop in which two cases occur, see Figure 7.3: either the end point of the edge lies somewhere inside the triangle (reached edge end), or the projected edge has intersected with one of the edges of the triangle.

In both cases a new node is generated for the three dimensional skeleton. In the first case the next edge is retrieved and processed. In the second case the algorithm moves to the triangle that shares the intersected edge and continues until the end of the edge is reached. A detailed description of the wrapping algorithm can be found in section **Error! Reference ource not found.**.

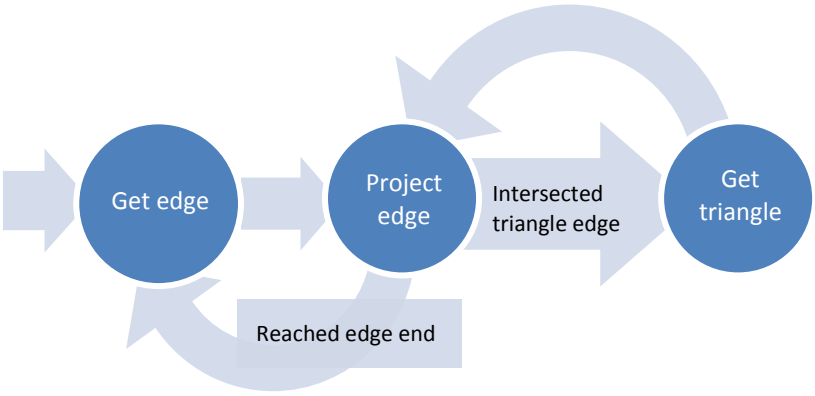


Figure 7.3 High level overview of the wrapping step.

Once the entire pattern has been processed the carving volume can be generated. This is essentially a two step process which consists of filtering operations and the actual mesh generation, see Figure 7.4. In the first step, filtering operations are performed upon to the skeleton to ensure the generated volume does not intersect itself, as this would cause problems during the Boolean operation. After the entire skeleton is filtered, it is processed and the carving mesh is generated. A more detailed description of the filtering steps and the carving mesh generation can be found in sections **Error! Reference source not found.** and REF _Ref261704059 \r \h **Error! Reference source not found.**.

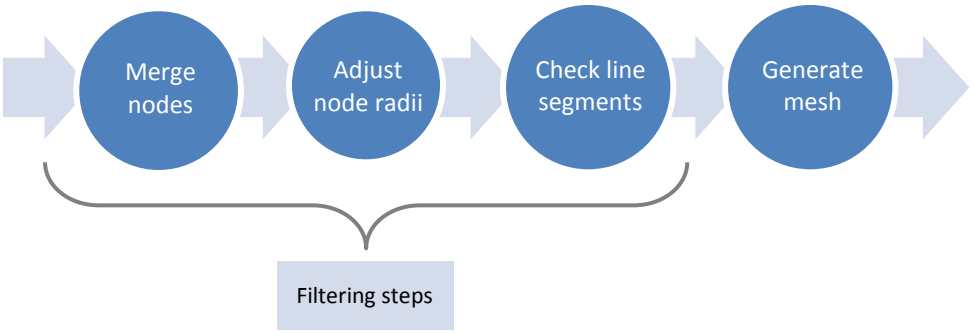


Figure 7.4 High level overview of the volume generation step.

7.3 Fracturing

Compared to the cracking sub-routine, the fracturing sub-routine is a lot simpler. The first two steps involve the generation of the fracture mesh from the fracture pattern and the calculations of the principal axis of the target model, see Figure 7.5.

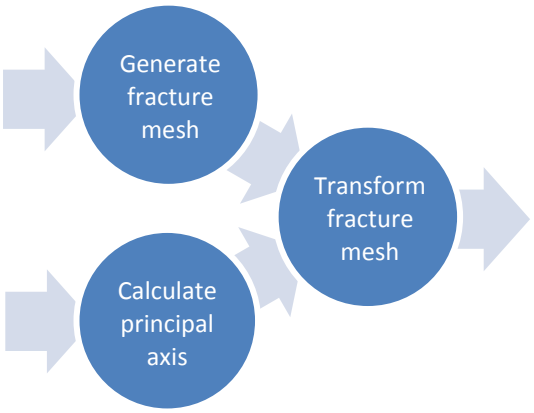


Figure 7.5 High level overview of the fracturing algorithm.

The fracture mesh is generated in a straightforward way; by simply extruding the line representing the planar fracture pattern in a direction perpendicular to its plane, see Figure 7.6. The geometry is generated in a unit size cube, so that the fracture mesh geometry can be reused by simply applying a transformation to the model for the specific fracture

instance, without needing to generate geometry multiple times for the same fracture pattern.

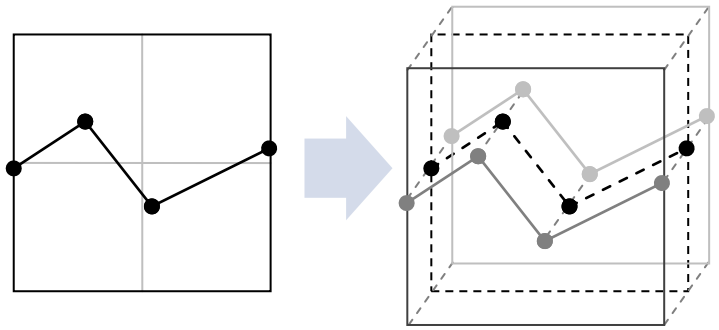


Figure 7.6 Fracture pattern extruded to a three dimensional fracture mesh.

The principal axis calculations result in 3 vectors which can be used as the up, right and forward vector of the transformation basis for the fracture mesh. The size of the target model can be used as a scaling factor for the fracture mesh, to ensure the fracture will fully intersect the target model. The fracture parameters can also be combined into the transformation. The rotation parameter can be included as a rotation component around the forward vector of the fracture mesh. The volume ratio parameter can be included as a translation along the up vector. After combining all these parameters into a single transformation matrix, the transformation is applied to the fracture mesh and it is ready to be used in the Boolean operation.

As we make sure that the fracture mesh fully intersects the target model, the fracture can be considered as splitting the model in two half-spaces, see Figure 7.7. One part of the target model will be in the top half-space, and the other part in the bottom half-space. The top part can then be constructed using a Boolean difference operation, while the bottom part can be constructed using a Boolean intersection operation, as the bottom half space represents the 'inside' of the fracture, see Figure 7.7. A more detailed description of the fracturing sub-routine can be found in subsection **Error! Reference source not found.**.

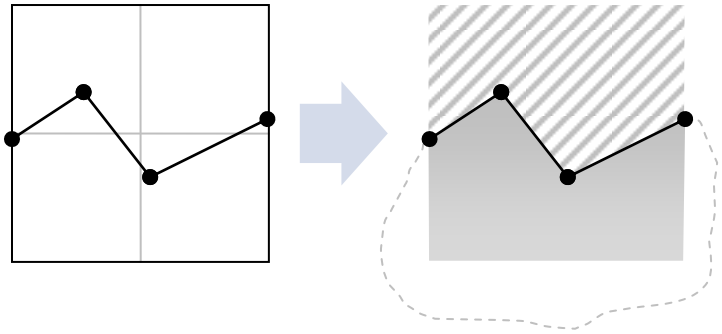


Figure 7.7 The fracture pattern forms two half-spaces. The bottom half-space can be considered to be the 'inside' of the fracture mesh.

7.4 Boolean operation

The Boolean operation between the destruction mesh and the target model is fairly straightforward and consists of 6 steps, see Figure 7.8.

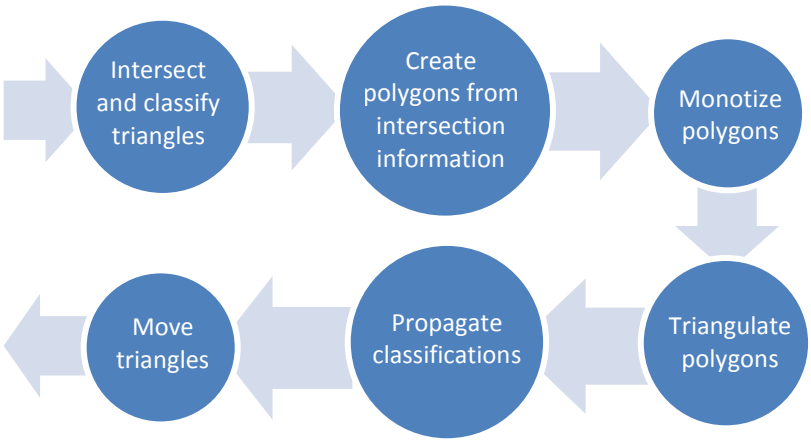


Figure 7.8 High level overview of the Boolean operation algorithm.

First, all intersections are calculated between the triangles of both models, storing all line segments that result from these intersections and determining the classifications, whether they are inside/outside the other model. Subsequently, this classification information is processed to create polygons, which are then processed to ensure they are all monotone so they can be easily triangulated in the next step. Once all polygons have been triangulated, the classifications that where determined earlier need to be propagated to the triangles which did not have any intersections, as these triangles will have an unknown classification at this point. In the final step, triangles are moved between the two models or removed, depending on the type of Boolean operation. A more detailed description of the Boolean operation can be found in subsection **Error! Reference source not found.**.

8 Implementation

This chapter has been removed because of intellectual property protection. Inquiries about this content can be directed at Cannibal Game Studios:

Cannibal Game Studios

Molengraaffsingel 12-14

2629 JD Delft

The Netherlands

contact@cannibalgamestudios.com

9 Results

With the algorithms and editors implemented as described in the previous three chapters, we can now present the results that can be achieved using the developed approach. Section 9.1 will show how the prototype can be used to design a destructible material, followed by section 9.2 where we will show how the prototype can be used to preview the created destructible behaviour. In section 9.3 result of the performance will be presented.

9.1 Designing a destructible material

The first steps in designing a destructible material are the creation of crack and fracture patterns, as described in section 5.1. In sections 9.1.1 and 9.1.2 we will show how crack and fracture patterns can be created. In section 9.1.3 we will show how these patterns are then used to create a destructible material.

9.1.1 Creating a crack pattern

When the user opens the editor to create a crack pattern, he is presented with an empty canvas, see Figure 9.1 (a). Nodes can be added to the pattern by holding shift and clicking. The currently selected node is highlighted in red, and, as can be seen in Figure 9.1, each new node is automatically selected so a sequence of connected nodes can be created without any additional selections having to be made.

To help identify the root node of the pattern, i.e. the point at which the crack originates, it is coloured differently from the other nodes in the pattern. Once a node has been created its settings can be edited, which will influence the shape of the volume that will be generated later. To indicate the approximate shape of this volume, the editor visualizes these settings in blue, as seen in Figure 9.1.

To better emulate the dissipation of impact energy the further away from the centre of the crack, the settings of nodes can be edited; the visualization is updated to reflect the changes, see Figure 9.2 (a). Should the user accidently input any erroneous values, the visualization will instantly alert the user to his mistake, see Figure 9.2 (b).

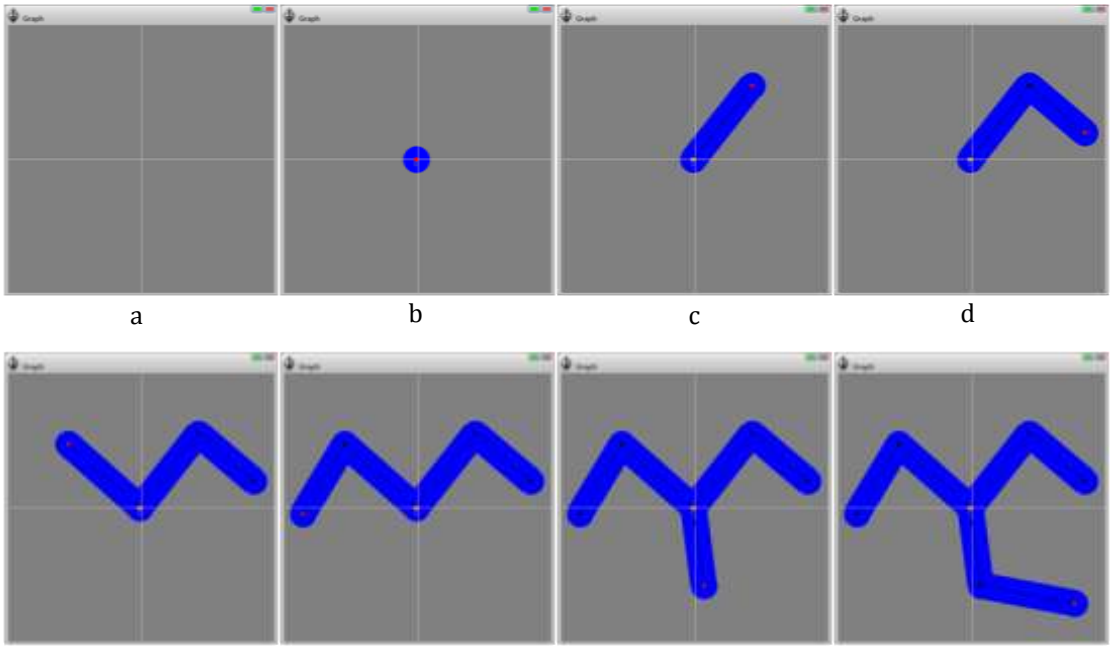


Figure 9.1 Creating a simple crack pattern.

The pattern shown in Figure 9.1 and Figure 9.2 is relatively simple, and was created in less than a minute. More complex patterns can be created in a similar fashion, a pattern as the one seen in Figure 9.3 can be created in less than five minutes.

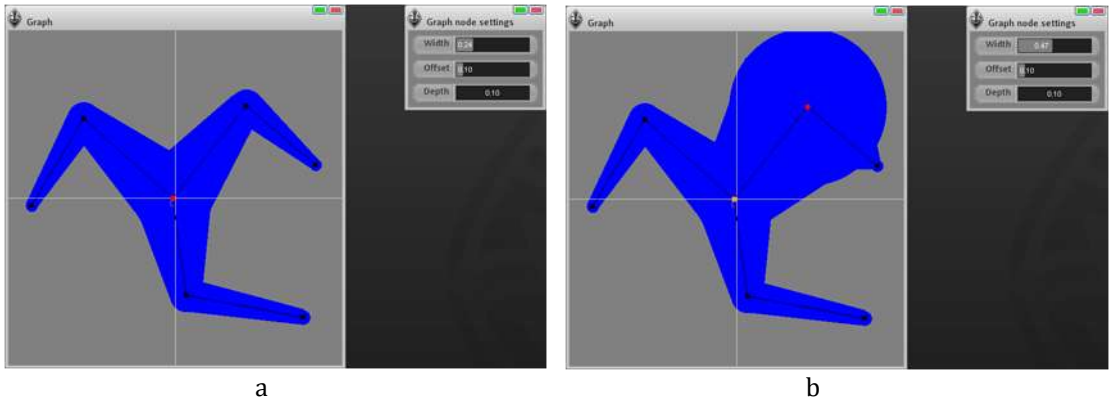


Figure 9.2 a: After creation node properties can be edited. **b:** Erroneous input values can be seen instantly.

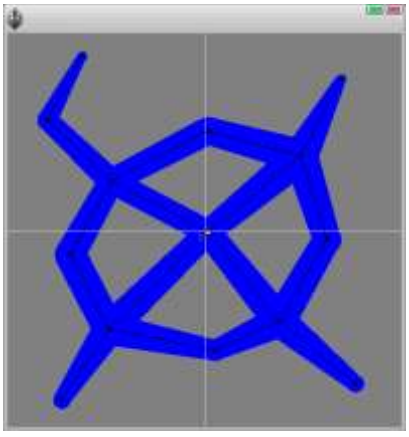


Figure 9.3 More complex crack patterns can also be created.

9.1.2 Creating a fracture pattern

When the user opens the editor to create a fracture pattern, he is presented with the default fracture pattern, see Figure 9.4 (a). Similarly to editing crack patterns new nodes can be added by repeatedly shift-clicking, as seen in Figure 9.4 (b) through (e). All nodes can be positioned using a single mouse stroke during which the two nodes nearest to the mouse are adjusted. After this initial placement the position of individual nodes can also be adjusted by clicking directly on them and then dragging them up or down.

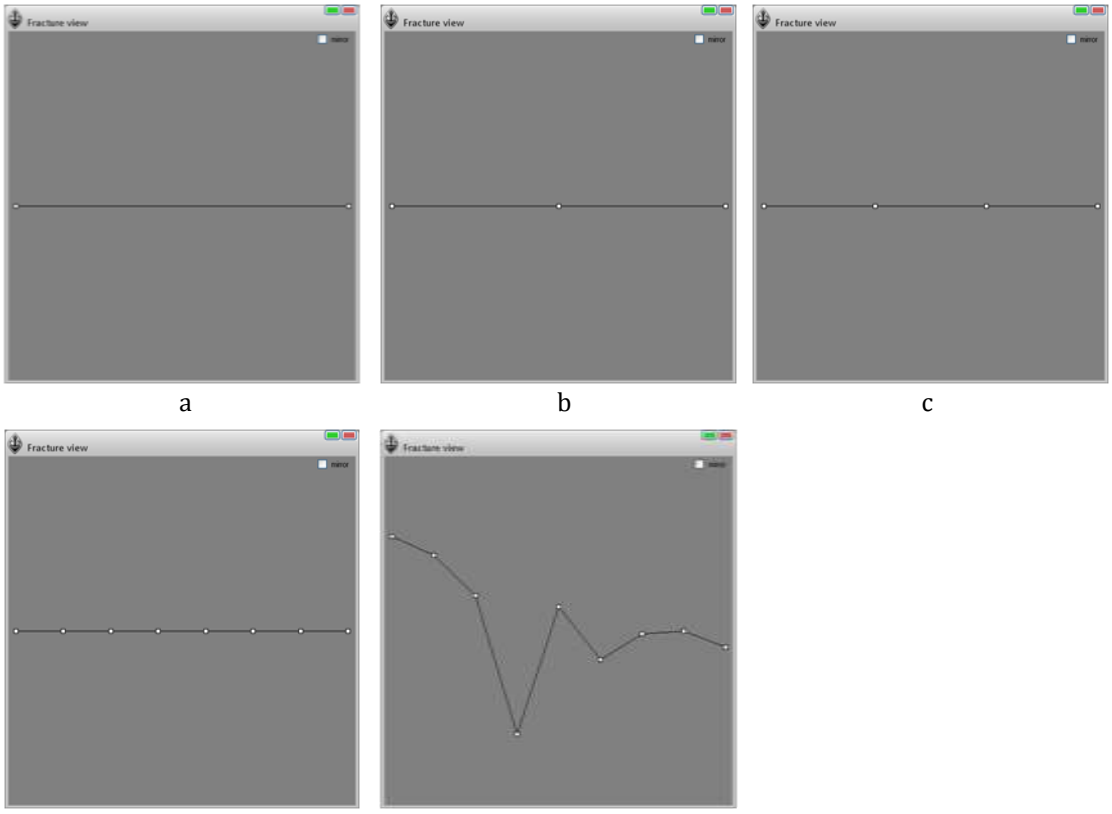


Figure 9.4 Creating a fracture pattern.

9.1.3 Creating a destructible material

When the user opens the editor to create a destructible, he is presented with several widgets, as can be seen in Figure 9.5. The previously created crack and fracture patterns are visible in the library widget, from which the patterns of choice can be dragged to the list view assigning them to the material. Once assigned in this manner, crack and fracture patterns can be selected and their settings edited, see Figure 9.6. To help identify specific patterns, any pattern that is selected is displayed on the canvas.

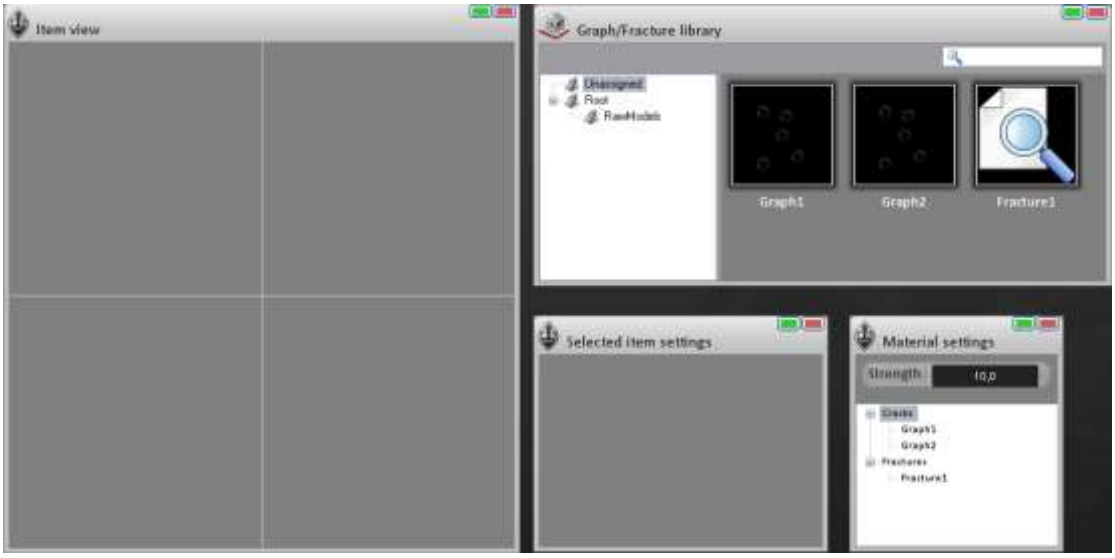


Figure 9.5 Patterns can be assigned to a material by dragging them from the library to the list view.

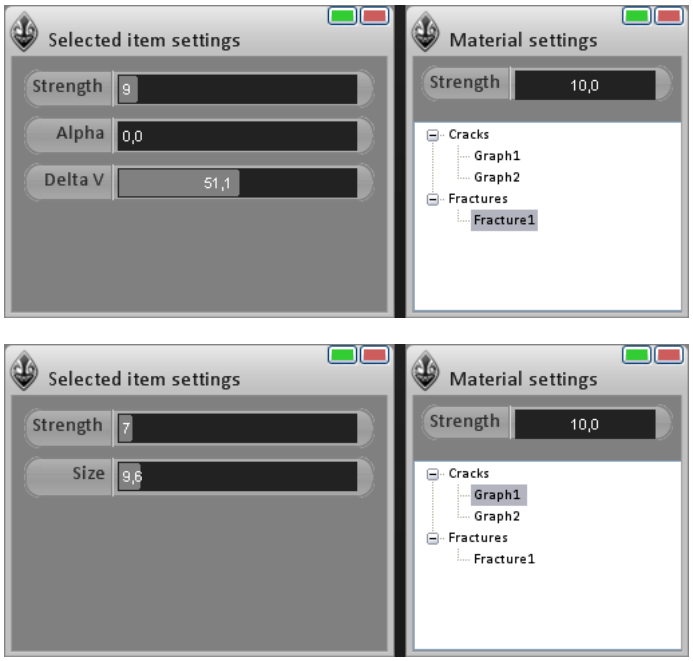


Figure 9.6 The settings for individual crack and fracture patterns can be edited by selecting them in the list view.

9.2 Previewing a destructible material

As soon as a pattern has been assigned to a material the destructible behaviour that has been created can be previewed. The user is presented a viewport, a settings widget and a library of objects, as seen in Figure 9.7. Three models have been loaded into the editor which can be used for testing the destructible material. In sections 9.2.1, 9.2.2, 9.2.3 and 9.2.4 we will show how these models can be used to preview the destructible material that has been created.

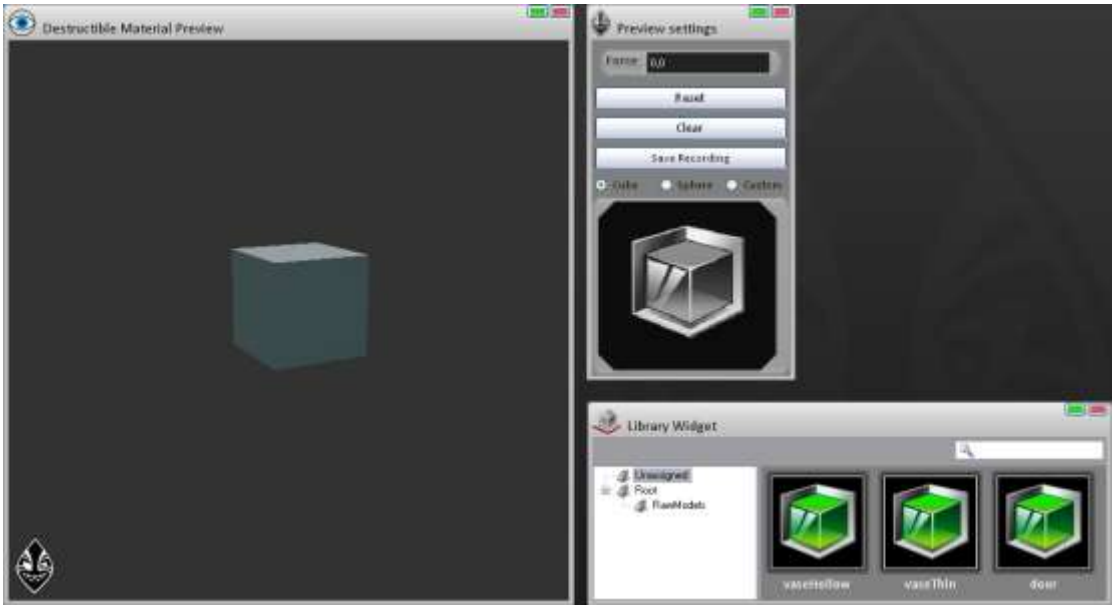


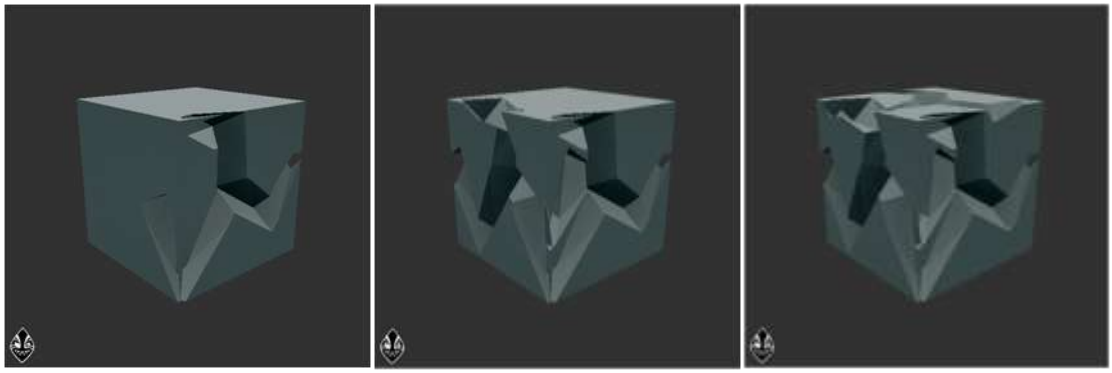
Figure 9.7 To preview a destructible material the user is presented a simple interface.

9.2.1 Cracking

To initiate a cracking operation, the force applied has to be lower than the strength of the material that was set during the design phase. In the prototype, the exact magnitude of the force does nothing else than determine if a crack or fracture operation should be performed. Once the user clicks on the viewport, as described in Section 6.3, a ray is cast to determine the exact triangle and position to place a crack pattern at. A crack pattern is then selected semi-randomly, as described in Section 6.1. With all this information a cracking operation is then performed as described in Section 7.2 and once the operation completes the resulting cracked object is displayed, as seen in Figure 9.8 (a).

Figure 9.8 New cracking operations can be performed on the result of previous operations.

More cracking operations can be performed on the result of the previously operation, see Figure 9.8 (b) through (e), allowing a user to simulate how their material would behave in a run-time environment. Should the user not unsatisfied with the results, he can simply switch back to the editing mode, change some settings like the size of the crack pattern, switch back, and evaluate the new results, see Figure 9.9.



a b c

Figure 9.9 After changing settings of the destructible material the new settings can be just as easy previewed.

However, due to stability issues of the implementation of the algorithms, sometimes the cracking operation fails to complete successfully. The most common errors occur during either the generation of the cracking volume, or during the Boolean operation between the cracking volume and the target object. In the case that the error occurs during the volume generation, nothing can be done other than informing the user of the failure. In the case of the error occurring during the Boolean operation we can visualize the cracking volume, see Figure 9.10, in order to help ascertain if there could be an obvious reason why an error would occur, for instance if the generated volume is irregularly shaped. In either case, the user has the option of saving the actions he performed by pressing the 'Save recording' button that was added to the settings widget, see Figure 9.7. This recording can be used as input for unit testing, so that once the source of the error is located and resolved we are assured that the same particular error will never return.

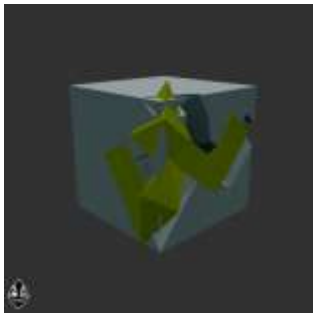


Figure 9.10 To help debug purposes the cracking volume is visualized should the cracking operation fail to complete successfully.

9.2.2 Additional cracking examples

Any custom model loaded into the editor can also be selected to preview the destructible behaviour. These models represent simple objects commonly found in games, namely a door and a vase, see Figure 9.11. The door consists of 208 triangles, for the vase there are 2 different versions. The first one which is solid, the way a vase would typical be modelled in a game, without an actual thickness, consisting of 288 triangles. The second version of the

vase represents a vase as it would be in real life, with an actual thickness, this version consists of 576 triangles.

Figure 9.11 The three custom models in their pristine state.

Figure 9.12 shows the door being cracked multiple times; a couple of artefacts can be seen like some polygons not being removed.

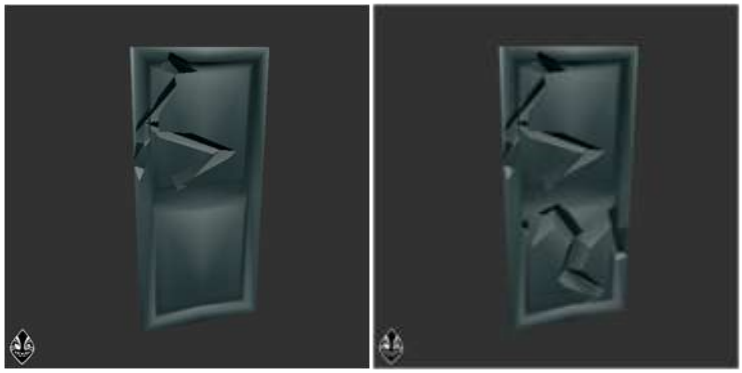


Figure 9.12 Door being cracked.

Figure 9.13 shows the solid vase being cracked multiple times. The volume removed by the cracking is entirely closed, giving the impression that the vase is a solid object.

a b c

Figure 9.13 A solid vase represented by a single shell, cracked repeatedly.

Figure 9.14 shows the thin vase being cracked. Because this vase has a thickness cracking it results in holes being created in the object and allowing us to see the insides. If the vase is

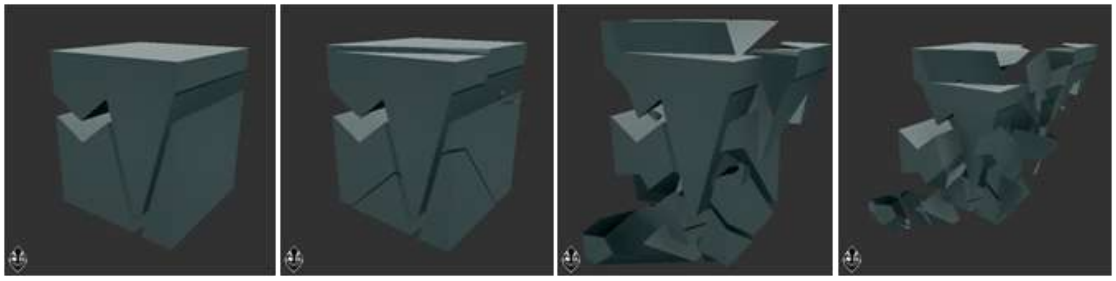
cracked on opposite sides, see the rightmost image in Figure 9.14, it is possible to see the background through the vase.

a b c

Figure 9.14 A thin vase with a modeled interior cracked repeatedly. In the right most image you can see through the vase after it is cracked on both sides.

9.2.3 Fracturing

Previewing fracturing is very similar to the way cracking is previewed. Once a user clicks on the object, and the force is high enough, a fracture pattern is randomly selected from the assigned patterns to apply to the object. All settings are applied and the target object is fractured once with the selected fracture, once the operation is complete the original model is removed and replaced with 2 fragments, see Figure 9.15 (a). Successive clicks apply a fracturing operation on all fragments created during the previous operation, as many times as desired by the user, as can be seen in Figure 9.15 (b) through (d).



a b c d

Figure 9.15 Successive fracture operations are applied on the fragments generated earlier.

Similar to cracking, stability issues sometimes prevent the operation from completing successfully, especially if a fragment has been fractured multiple times already. Any fragments which cannot successfully be fractured are then simply no longer included in the set of fragments to fracture should a user perform another destruction operation.

9.2.4 Additional fracturing examples

Similar to the additional cracking examples all three models can be fractured. Figure 9.16 shows the door model being fractured: in (a) and (b) the shape of the fracture pattern is

clearly visible, but in (c), after three iterations of the fracture operation, the fragments become less distinct. In (d) most fragments start to become unrecognizable pieces.

a b c d

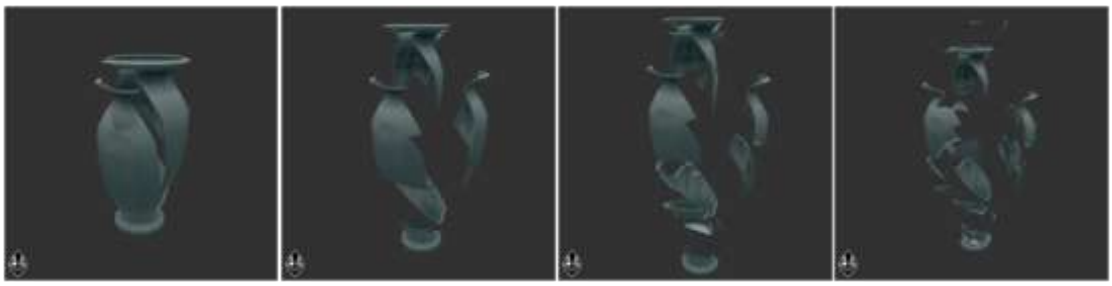
Figure 9.16 The door model being fractured.

Figure 9.17 shows the solid vase model being fractured. Similar to the door, the further the object is fractured the less distinct all fragments become. A drawback of using a 'solid' object to represent an object that is in fact hollow can be clearly seen. As the vase is considered to be hollow every fragment forms a closed surface, while a real vase would consist of thin pieces.

a b c d

Figure 9.17 The solid vase being fractured.

Figure 9.18 shows the thin vase being fractured. Contrary to the solid vase, when fracturing the thin vase flat fragments are created, similarly to how a real vase would break.



a b c d

Figure 9.18 The thin vase being fractured.

9.3 Performance

One important aspect of the prototype is the performance of the created algorithms, as the goal is to apply the presented approach in a real-time environment. In this section we will

therefore discuss some results regarding the performance of the prototype. All performance testing was performed on a normal desktop PC, with the following specifications; Intel core duo 6600 @ 2.4 GHz, 2 GB of RAM, Geforce 8800 GTX, Windows XP SP 3. In subsection 9.3.1 we will present some in-depth performance results, while in subsection 9.3.2 we will discuss performance scaling.

9.3.1 Performance per method

Two test cases were created from which detailed performance information was recorded. The solid and thin vases where used to perform a cracking operation, creating a result similar to that in Figure 9.13 a) and Figure 9.14 a). In these test cases, the methods involved with the graph application, mesh generation and Boolean operation where timed. To time each method call, a built in time measurement system of the Cannibal Engine was used. Each measurement records the time between calls to a start and stop method, like the standard .Net stopwatch, and for each frame the total time is then stored. While this system will not provide highly accurate results and introduces some overhead, it will provide a good indication of the performance. For each test the cracking operation was performed 100 times in order to get a good average result.

The performance results for the solid vase can be seen in Figure 9.19. In both test cases, the graph application and mesh generation methods only required, on average, zero to two milliseconds to complete, therefore only the timings for the Boolean operation methods are significant. The figure shows that the triangle intersection method on average takes fifty percent of the time of the whole cracking operation. Twenty five percent is needed to move triangles between the models; the remaining twenty five percent is taken by the rest of the methods, like propagating classifications and updating the boundary representation, as well as the earlier steps of the process like the graph application and mesh generation. Two large outliers can be seen, which run off the chart, as well as other small peeks. These two large peeks are most likely caused by the garbage collector being triggered while the cracking operation is being performed. The remaining other peeks are likely caused by external factors, like threading.

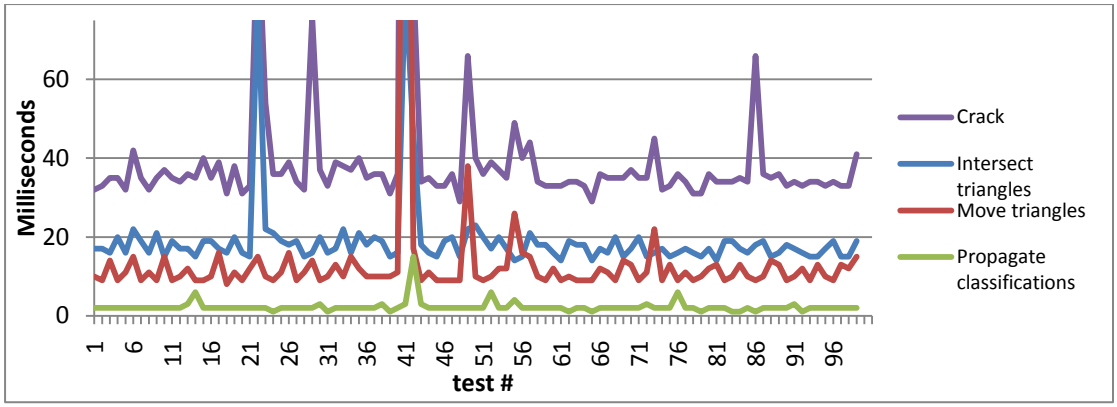


Figure 9.19 Performance measurements for the solid vase test case.

Figure 9.20 shows the performance results of the thin vase test case, which shows the exact same patterns as the solid vase test case. The main difference in these timings is that they are slightly higher, most likely because the thin vase model consists of more triangles. Once again two large outliers can be seen.

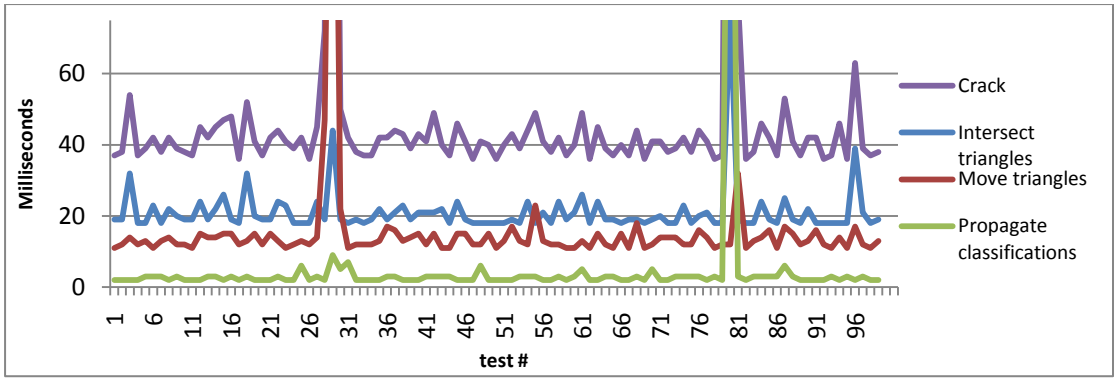


Figure 9.20 Performance measurements for the thin vase test case.

It should be noted that the timings reported here were gathered in debug mode, due to the timing mechanism being disabled in release mode. To get an idea of the timings for a release mode build, a single timer was implemented by hand to time the entire cracking operation. Figure 9.21 shows the recorded timings of a release mode build of the thin vase test case, showing a 50% decrease in required time to complete the operation. Also, no large outliers were detected, most likely due to the reduced number of debugging operations and a lack of debug output which results in less memory being used.

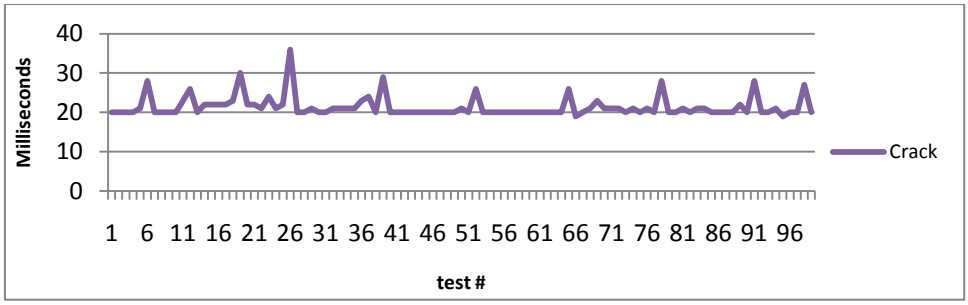


Figure 9.21 Reported cracking timings for release mode.

9.3.2 Performance scaling

To gather more information on the scaling of the performance when the model complexity increases, timing information was gathered for each cracking example in Sections 9.2.1 and 9.2.2, in a similar fashion as the timings reported in Figure 9.21. Figure 9.22 would suggest that the cracking algorithm is roughly O(n). However it is very hard to accurately determine the performance scaling as the algorithms in the different steps of the destruction algorithms have different orders of complexity.

For instance, cracking the door model took relatively long, compared to the 'cube small graph' example. This however, is simply caused by the space of the object being cracked. In the case of the door, which is high, wide and flat object, triangles on both sides of the model will be found to be near the carving volume and will have to be submitted for detailed intersection checks. As the triangle-triangle intersection is probably around $O(n^2)$, or something similar, performing a crack operation in an area with many triangles will be more detrimental to the performance than the object having a high global triangle count. This can be further seen in the 'cube small graph' example, where the last 2 cracking operations take nearly the same amount of time as the one before those, due to the cracking operation being performed in an area with few triangles.

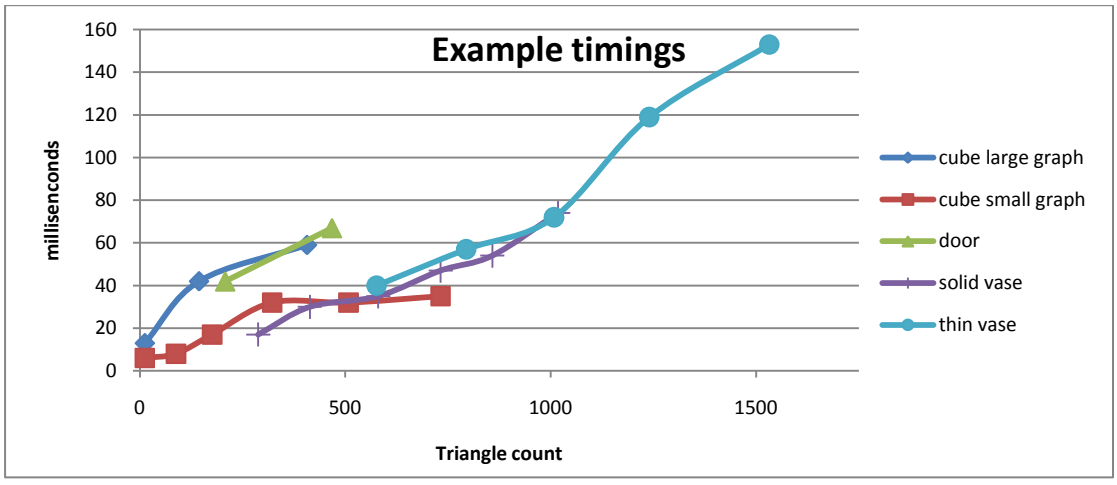


Figure 9.22 Release mode timing results of the cracking examples.

10Conclusions

The goal of this project was to investigate how to create a designer friendly approach for authoring specific destructible behaviour which can then be applied to a variety of objects in real-time.

A survey of research in this area showed that no existing approach satisfies enough criteria, like ease of use, user control or speed, as specified by industry professionals. Approaches that could be applied in real-time would usually be very abstract or lack user control over the generated result, or vice versa.

Looking at commercial solutions it becomes clear that even though they can create some spectacular results, they often still require a lot of manual labour. The use of the specific software is usually also limited to a handful of games created by the same studio that developed the software.

As basis for the approach developed in this thesis, an existing method was used that satisfied the criteria of user friendliness. The novel concept of destructible materials has been introduced to extend that basic approach with reusability, making it more designer friendly and allowing it to be easily applied to different objects. We can therefore say that the first goal has been successfully achieved.

The prototype implementation proved that destructible materials enable designers to easily create reusable destructible behaviour which can be applied to a variety of objects. Its current implementation also showed that for simple cases this destructible behaviour can be applied in real-time. Therefore, we can consider the second goal to be, at most, partially achieved.

The presented work focuses largely on the technical aspect of destructible behaviour, and while it demonstrates the feasibility of the approach it does not present a finished work. A lot of work remains, some on the technical side, but also on the user experience side. However, the presented work does provide a step in the right direction.

10.1 Recommendations

While this project is now completed, there are still various aspects of the prototype system that could be improved, which will now be briefly discussed.

Editor

While the editor enables designers to create reusable destructible behaviour in a fast and easy manner, there are several ways in which the editing process could be further improved:

Currently the editor uses a very old-school point-and-click style interaction paradigm; this paradigm could be made a lot more dynamic by actually allowing a user to draw a crack pattern which is automatically converted into a graph.

This could even be further extended by enabling users to specify cracks directly on the surface of an object from which a two dimensional crack pattern is created and stored for later reuse.

Algorithms

The prototype implementations of the described algorithms showed that the destructible behaviour defined by using the presented approach can be applied in near real-time. The majority of the examples displayed took less than eighty milliseconds to complete, which is only five frames at sixty Hz. This is a very promising result considering the implementations of the algorithms are the part of the prototype where the largest gains can be achieved in future optimization work.

Currently the largest problem areas are the stability of the system and the computation complexity. The stability issues are largely related to problems resulting from floating point inaccuracies, but the sheer complexity of the code base and algorithms makes it difficult to pinpoint the source of a lot of problems. Programming errors in one section of code often do not create a problem until later, greatly increasing the difficulty of finding the source of the problem.

The triangle-triangle intersection calculations are by far the most computationally expensive part of the prototype. However, there are many different things that can be done to reduce this cost. In the prototype, the intersection calculations are all performed on a single thread. By using a multithreaded approach, or even a hyper-threaded approach like a GPU-based solution, the time needed to calculate all intersections could be massively reduced. Not only can we reduce the computational cost on a per operation basis, more efficient broad phase filtering, using a more efficient spatial data structure, could further reduce the CPU load.

Process

One aspect of this thesis project that should not be forgotten is the process itself. In hindsight the amount of work required for implementing the prototype system was vastly underestimated. As a direct result of this, many aspects of the prototype system did not receive as much attention as possible as much time was spend on simply getting everything to work. As a side effect, not only did the project go way overdue, it also caused a strain on moral which in turn only created further delays. In future projects, more care will have to be taken in creating a planning, and thought will have to be put into creating a plan that has contingencies for the case in which the project does go overdue.

My recommendation to Cannibal Game Studios is to further develop the present approach into a fully-fledged product. While a lot of work needs to be done in order to achieve this, the presented approach is a technology that would provide a unique selling point and shows that the company is able to take existing ideas to new levels.

Bibliography

1. **LucasArts.** Star Wars: Force Unleashed - Digital Molecular Matter Tech Demo. [Online] February 14, 2007. [Cited: January 5, 2009.] http://www.gametrailers.com/player/17054.html.

2. **Nealen, Andrew, et al.** Physically Based Deformable Models in Computer Graphics. *Computer Graphics Forum.* 2006, pp. 809-836.

3. **O'Brien, James F and Hodgins, Jessica K.** Graphical Modeling and Animation of Brittle Fracture. *SIGGRAPH.* 1999, pp. 137-146.

4. **Müller, Matthias, Teschner, Matthias and Gross, Markus.** Physically Based Simulation of Objects Represented by Surface Meshes. *Computer Graphics International.* June 2004, pp. 26-33.

5. **Galoppo, Nico, et al.** Fast simulation of deformable models in contact using dynamic deformation textures. *Symposium on Computer Animation.* 2006, pp. 73-82.

6. **Müller, Matthias, et al.** Meshless Deformations Based on Shape Matching. *ACM Trans. Graph.* 2005, Vol. 24, 3, pp. 471-478.

7. **Rivers, Alec R. and James, Doug L.** FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation. *ACM Trans. Graph.* 2007, Vol. 26, 3, p. 82.

8. **Martinet, Aurélien, et al.** Procedural Modeling of Cracks and Fractures. *SMI.* 2004, pp. 346-349.

9. **Fox, Daniel.** *Demolishing Objects in Computer Games.* 2003. Msc Thesis.

10. **Miller, Adam.** *A Cracking Algorithm for Exploding Objects.* 2004. Msc Thesis.

11. **Workman, Steven.** *A Cracking Algorithm for Destructible 3D Objects.* 2006. Msc Thesis.

12. **Smith, Jeffrey, Witkin, Andrew and Baraff, David.** Fast and Controllable Simulation of the Shattering of Brittle Objects. *Comput. Graph. Forum.* 2001, Vol. 20, 2, pp. 81-90.

13. **Havok.** Home. [Online] Havok. [Cited: June 23, 2010.] http://www.havok.com/.

14. —. Havok Destruction Product Brief. [Online] [Cited: March 10, 2010.] http://www.havok.com/uploads/Havok_Destruction_Brief_Mar%204.pdf.

15. —. Havok product showcase. [Online] [Cited: March 10, 2010.] http://www.havok.com/index.php?page=showcase.

16. **Pixelux.** Pixelux Entertainment. [Online] [Cited: June 23, 2010.] http://www.pixelux.com/.

17. **Nvidia.** PHYSX. [Online] [Cited: June 23, 2010.] http://www.nvidia.com/object/physx_new.html.

18. **Unreal.** Unreal Technology. [Online] [Cited: June 23, 2010.] http://www.unrealtechnology.com/.

19. **Epic Games.** Static Mesh Editor User Guide. [Online] 2008. http://udn.epicgames.com/Three/StaticMeshEditorUserGuide.html.

20. **Hubbard, Philip M.** *Constructive Solid Geometry for Triangulated Polyhedra.* 1990.

21. **Dice.** DICE. [Online] [Cited: June 23, 2010.] http://www.dice.se/.

22. **IGN.com.** Battlefield: Bad Company Xbox 360 Commentary - Developer Commentary. [Online] December 18, 2007. [Cited: March 11, 2010.] http://xbox360.ign.com/dor/objects/713943/dice-project-2/videos/bfbc_destruction_121407.html.

23. **GameTrailers.com.** Battlefield Moments Episode II. [Online] November 19, 2009. [Cited: March 11, 2010.] http://www.gametrailers.com/video/battlefield-moments-battlefield-bad/59221.

24. **Ericson, Christer.** *Real time collision detection.* 2005.

25. **Moller, Tomas.** A fast triangle-triangle intersection test. *Journal of Graphics Tools.* 1997, pp. 25-30. See http://jgt.akpeters.com/papers/Moller97/tritri.html (Last accessed April 2009).

26. **O'Rourke, Joseph.** *Computational Geometry in C.* s.l. : Cambridge University Press, 1998.

27. **de Berg, Mark, et al.** *Computational Geometry.* 2nd revised edition. s.l. : Springer-Verlag, 2000. pp. 45-61.

28. **Moondoggy.** Deformation (engineering). *Wikipedia.* [Online] May 13, 2008. http://en.wikipedia.org/wiki/Image:Stress-strain1.svg.

29. **Galoppo, Nico, et al.** Fast simulation of deformable models in contact using dynamic deformation textures. *Symposium on Computer Animation.* 2006, pp. 73-82.

30. —. Dynamic Deformation Textures, GPU-Accelerated Simulation of Deformable Models in Contact. *SIGGRAPH courses.* 2007, pp. 59-79. Advanced Real-Time Rendering in 3D Graphics and Games Course.

31. **Allègre, Rémi, et al.** A Hybrid Shape Representation for Free-form Modeling. *Shape Modeling International.* 2004, pp. 7-18.

32. —. A Hybrid Shape Representation for Free-form Modeling. *Shape Modeling International.* 2003.