Modelling Objects for Simulation of Breakables

Jens-Kristian Nielsen

Kongens Lyngby 2013
IMM-M.Sc.Eng.-2013-28

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.Eng.-2013-28

Abstract

In English

This thesis describes a destruction system, which has been developed to create and simulate breakable objects in a real-time gaming context. The goal of this thesis is to describe how this destruction system, which can be used in a low-budget game development, was created. The design of the destruction system is based on a study of related works that appear in graphics literature. Details concerning implementation of how fragments are procedurally modelled and how destruction is simulated are explained. Finally the system is tested to determine its limitation and the results are reflected upon.

In Danish

Dette speciale beskriver et system, der er udviklet til at skabe og simulere objekter, der kan gå i stykker i realtid i spil. Målet med denne afhandling er at beskrive, hvordan et ødelæggelses-system, der kan bruges i lav-budgets spiludviklingsprojekter, er blevet udviklet. Ødelæggelses-systemets design er baseret på en undersøgelse af relaterede værker, der findes i grafiklitteraturen. Det forklares hvordan fragmenter proceduremæssigt bliver modelleret, og hvordan ødelæggelse er simuleret. Til sidst testes systemet for at finde dets begrænsninger, og der reflekteres over resultaterne.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc.Eng in Digital Media Engineering. It contains work done from October 2012 to March 2013. The thesis has been written under the supervision of Jakob Andreas Bærentzen of the Department of Informatics and Mathematical Modelling at DTU. The thesis documents the development of a destruction system for the Unity game engine.

To fully understand the contents of this thesis the reader must have a basic understanding of software development, computer science, three-dimensional computer graphics, C# programming and have some familiarity with game engines, rendering engines and rigid body physics engines.

Thesis Structure

Chapter 1: Introduction
This chapter describes project and defines the goal of the thesis.

Chapter 2: Related Works
This chapter describes and compares various methods that others have used to simulate destruction in real-time. I make simple implementations of some of these methods to discover which works best within the constraints of Unity.

Chapter 3: Overall Design

This chapter describes the general concept of how the destruction system works and why it was designed this way.

Chapter 4: Modeling Fragments
This chapter describes how the fragments are procedurally modelled.

Chapter 5: Simulation
This chapter describes how the breakage is simulated.

Chapter 6: Tests and Results
This chapter describes various tests used to determine how well the destruction system works, and what its limitations are.

Chapter 7: Conclusions
This chapter reflects on the process and the results, discusses hindsights and how the destruction system could be further improved.

Appendix A: User Manual
This appendix describes how to use the destruction system to produce destructible objects in a game.

Copenhagen, Denmark, 29-March-2013

Jens-Kristian Nielsen

Jens-Kristian Nielsen

Acknowledgements

I would like to thank my supervisor, Jakob Andreas Bærentzen. His ideas, inspiration, comments and his endless patience at our weekly meetings, some of which extended far beyond the schedule were essential to the success of the project.

Thanks to Kenny Erleben of DIKU, University of Copenhagen for spending time emailing and meeting with me to discuss the project.

Finally thanks to Kassandra Flindt for her patience and understanding, and to my proof readers: Stine Rosenbeck, Asbjørn Clemmensen, Kristian Harving, Frederik Buus Sauer, Anne Elvira Enkelund, Rasmus Kann Jensen, Valdis Vilcan, and Lelia Peuchamiel for helping me make this thesis a little more readable. And anyone else who has supported me throughout the process.

Contents

Chapter 1

Introduction

Destructible objects and environments have been a widely used feature throughout the history of video games. For instance Space Invaders (1978) featured destructible bunkers which the player would be able to use as cover and which would gradually be destroyed by both the enemies and the player. Similarly, in the game Asteroids (1979), the main objective was to destroy asteroids bit by bit.

Modern 3D games like the ones in the Battlefield franchise and the Red Faction franchise have also featured destructible environments in the form of deformable terrain and destructible buildings. Even games that are not centered around destruction may feature some form of destructible objects. For instance destructible cover in the shooter genre is becoming increasingly popular. Another example of how important destruction is to games is the international 2D hit Angry Birds in which the player must destroy a construction by catapulting birds at it.

In summation, there is continuous need for simulating breakables in games as they often constitute a defining element of the game experience. In this thesis i propose a tool that makes this feature more accessible to game developer making low-budget games.

1.1 Project Description

In many cases destruction is not very dynamic. Canned animations (animation premade by artists that are played on certain events) only have limited use as gameplay elements and are only useful as visual effects. In many cases, however games do not feature destruction at all, even when it might be appropriate in the games context.

The technology for physically based animation has made major progress through the last few decades, and combined with the continued progress of processing power due to Moore's law we have reached a point where dynamic realtime destruction can be implemented in a wide range of games. The last few years have produced many such systems, but it remains almost exclusively in big-budget titles like the previously mentioned Battlefield and Red Faction franchises. It has yet to be widely available to indie, mobile device and student developers, because destruction systems are often elaborate and expensive to produce.

1.2 Goal

The purpose of this project is to make it easy and fast for developers to create destructible objects in their games. This contribution will assist the lower-budget productions in reaching a higher production value and subsequently help bridge the chasm between the triple A titles and the smaller indie productions. In this way the thesis attempts to widen the scope of possibilities for low-budget game producers, hoping that more innovative games will see the light of day. This is achieved by creating a destruction system for the game engine Unity, which is widely used by various game developers namely indie, mobile device and student developers. To be able to implement this system within the thesis deadline, the product is limited to only being able to fracture rigid, non-deformable materials. Materials that could be simulated in this system are for instance porcelain, stone, concrete and glass.

Chapter 2

Related Work

In this chapter I study various related work to determine whether any current solutions or research can help me create a system that can produce realistic-looking and dynamic breakable objects, thus prevent me from having to reinvent the wheel.

2.1 Comparison of Methods

At the Siggraph conference in 2011 Erwin Coumans [Cou11] compared some of the different methods currently used in destruction systems, which he later elaborated on in a post on #AltDevBlogADay [Cou12]. This chapter primarily is based on those sources.

Most destruction systems are divided into two parts, which do the following:

- Process the 3D geometry to produce fragments. This is done in a before runtime.
- Simulate the breakage at run time.

This general approach is referred to as *preshattering*. When one wants to incorporate destruction in game and film productions it is normally considered more important that it looks realistic and that it is computationally cheap than that it truly acts physically realistically. This is why preshattering is such a popular model for destruction systems. It is much cheaper to mimic destruction by having a limited number of pieces falling apart than to actually determine how an object divides into pieces based on its physical properties at run time. In a game environment, maintaining a framerate is essential, and allowing the number of fragments to grow without limit is undesirable.

In table 2.1 commonly used approaches to each of the two parts of destruction systems are listed.

Geometry Preparation	**Runtime Destruction**
Manual modeling	Canned Animation
Boolean Operations	Real-time Booleans
Cut-Out	Connected Particles
Tetrahedralization	Finite Element Method
Convex Decomposition	Breakable Constraints
Voronoi Shattering	Breakable Composite Rigid Body
Cutting/Slicing	-

Table 2.1: Geometry preparation and simulation methods for destruction systems.

It is worth noting that some methods, both for geometry preparation and simulation, are good at imitating certain material properties and bad at imitating others. Some geometry preparation methods produce fragments that have clean surfaces, which are appropriate for materials like minerals, porcelain or crystals and some simulation methods are good for rigid, non-flexible destruction, which goes well with the aforementioned materials. But those same methods might be inappropriate for materials like wood which break into splintery pieces and is flexible, which means that it bends before it breaks.

2.1.1 Geometry Preparation

In a preshatter destruction system, the object that is to be made destructible needs to be divided into pieces, so those pieces can fall apart during the simulation. These pieces will be referred to as fragments. This section describes some of the popular methods that are used to divide a 3D model into fragments. When studying the different geometry preparation methods, we are interested in the following qualities:

- Artistic control, meaning the artist or designer using the destruction system to produce the fragments has a high degree of control in how the pieces are shaped.
- The method is fast and easy to use.

We are however not particularly interested in the system being computationally fast since the geometry preparation step is not performed at runtime.

When producing fragments, one of the challenges is that we only have information about the shell of the object, since we are dealing with polygon meshes, so we must reconstruct the interior of the objects as we take them apart. The following methods have different ways of handling this issue.

2.1.1.1 Manual Modelling

The simplest way of producing fragments is to model them in a content creation tool like Autodesk Maya, 3ds Max or Blender. This is the method that allows the highest degree of artistic control, but since it not procedural it may be very costly in terms of man hours.

2.1.1.2 Boolean Operations

Boolean operations can be used to perform volumetric operations between two meshes. Computing differences or intersections can be used for mesh decomposition. They allow us to break a mesh into pieces, similar to using a cookie cutter. Instances of spatial booleans operations can be seen in figure 2.1. This

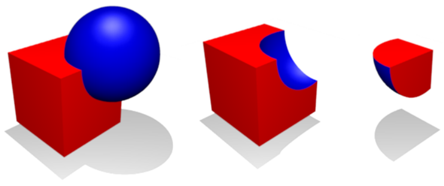
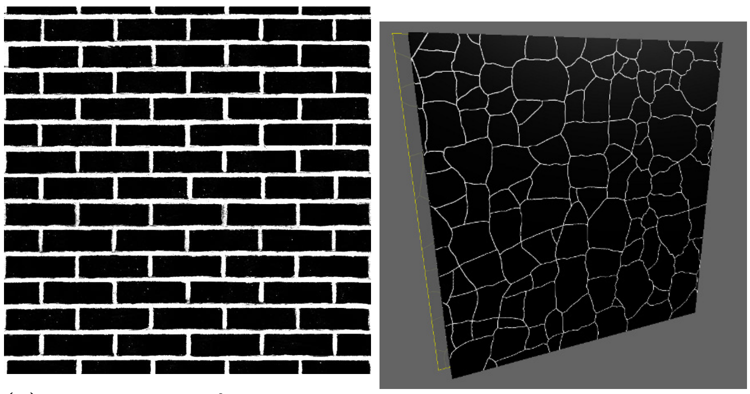


Figure 2.1: Boolean operations. Source: [Cou12]

is still a type of manual modeling, not procedural, so it relatively slow, but it gives a high degree of artistic control.

2.1.1.3 Cut-out

In this method an artist provides a *fracture map* with fracture lines as a bitmap file, which can be produced in content creation tools like Photoshop, by drawing white lines onto a black background. Figure 2.2a illustrates a fracture map for producing a brick-like fracture pattern. The fracture map is projected onto the object, as seen in figure 2.2b, and the mesh is decomposed based on those lines. This method provides a high degree of artistic control, but also requires a lot of manual work by an artist.



(a) Fracture map for a creating brick-like fragments.

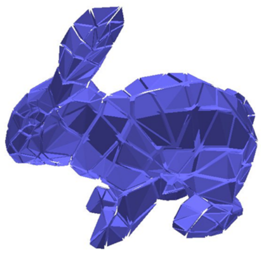
(b) Projected onto object.

Figure 2.2: Cut-out. Source: [u0910]

2.1.1.4 Tetrahedralization

Tetrahedralization is the process of dividing the source mesh into a set of tetrahedra as seen in figure 2.3a. This is accomplished using delaunay triangulation, which is closely related to voronoi regions. Creating tetrahedron shaped fragments is not necessarily very interesting because they do not in themselves look like natural fragments, but they are useful for simulation purposes because of their geometric simplicity as seen in figure 2.3b.

In such cases the tetrahedralized version is not used for visualization; this would have to be done using one of the other methods.


(a) Tetrahedralized Bunny.

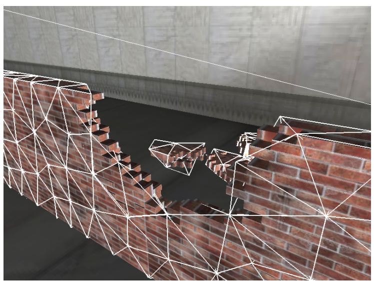

(b) Tetrahedron for simulation.

Figure 2.3: Tetrahedralization. Source: [Cou12]

2.1.1.5 Convex Decomposition

In the convex decomposition method a concave mesh is divided into smaller convex parts, which can be seen in figure 2.4. This can be done manually by using convex primitives or it can be done procedurally. Convex decomposition is a NP-hard problem, but it is possible to create approximations relatively cheaply. One approach to doing so is described in [MG09]. This methods breaks

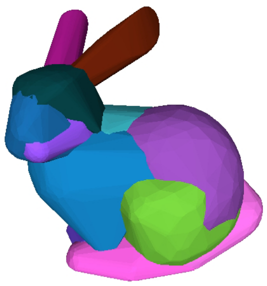
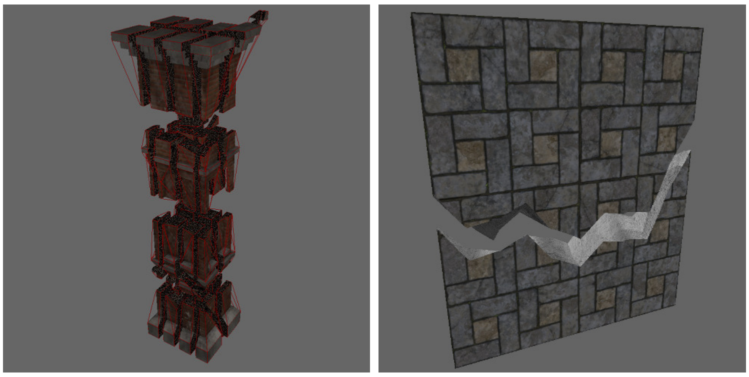


Figure 2.4: Convex Decomposition. Source: [Cou12]

the source mesh into fragments in a very natural way, for instance the bunny's ears break off, rather than break at some arbitrary point, but the number and shape of those fragments depend entirely on the source mesh and involves no artistic control. This method could in theory be useful if combined with another method, so the mesh would first be divided using convex decomposition and then those pieces would be divided using another method.

2.1.1.6 Slicing/Cutting/Splitting

The slicing method, which is also sometimes called cutting or splitting, is not really one specific method, but rather a group of methods which all involve dividing the source geometry using planes and then filling the resulting gaps. One popular approach, used in PhysX Apex Destruction, is to divide the source mesh a given number of times in the X, Y and Z axis, seen in figure 2.5a, then apply offset, angular variation and noise to those planes. This produces very realistic fragments and allows a high degree of artistic control. Figure 2.5b shows a single slice with offset, angular variation and noise applied.



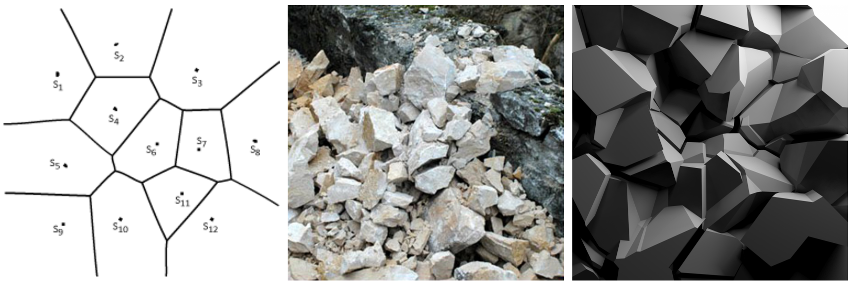
(a) A pillar sliced in multiple times in the X, Y and Z axis. **(b)** A wall segment sliced with plane distorted with noise.

Figure 2.5: Slicing. Source: [u0910]

2.1.1.7 Voronoi Shattering

Voronoi shattering is a procedural method based on voronoi diagrams, depicted in figure 2.6a. This approach is a way to divide space into regions, called voronoi cells. It is typically used to imitate materials like rocks or minerals as seen in figure 2.6b. The result of voronoi shattering can be seen in figure 2.6c.

[Est11] and [Led07] describes how voronoi shattering works. First the polygonal mesh is sampled to find a set of point within it called *seeds*. For each seed we create a set of planes that makes out its corresponding voronoi cell. For each of these planes the mesh is sliced and the resulting gap is covered with new faces. The result is a set of convex meshes corresponding to the fragments.



(a) Voronoi diagram. **(b)** Typical material. **(c)** Result.

Figure 2.6: Voronoi shattering. Source: [Est11]

2.1.2 Runtime Destruction

Once the source object has been divided into fragments we need to execute the destruction in the game. The quality we are mainly interested in in the runtime part of the destruction system is that it acts realistically, since we are mainly concerned with games, and that it is computationally cheap.

2.1.2.1 Canned Animation

Instead of doing a physically-based animation at runtime, the simulation can be done in the preparation step or it can be done as a manual animation in a content creation tool. At run time, the animation is simply played back when triggered. This is computationally very cheap and allows a great deal of artistic control. The disadvantage with this approach is that the object will always break as it was animated and it will not correspond exactly to the interaction performed in the game.

2.1.2.2 Real-time Booleans

Boolean operations can also be used at runtime, which means that this method does not require a geometry processing preparation step, thus is not a preshattering method. In a destruction system based on real-time booleans destruction works in the following way: When the environment is hit with great enough force, a shape that corresponds to the damage made is generated procedurally. Subsequently the intersection of the generated shape and the environment is removed from the environment geometry.

Level design can prove challenging when using real-time booleans since the player can perform a much greater number of modifications compared to pre-shattering methods. [BNT90] describes how to accomplish this by merging BSP trees. This approach requires additional analysis to determine if objects need to collapse.

2.1.2.3 Connected Particles and Finite Element Method

Destruction and fracture can also be simulated using particles. This approach is similar to rope physics where particles are connected into chains, two by two. Connecting particles four by four into tetrahedra allows deformation simulation of 3D objects and can also be used to simulate destruction by breaking these connections.

By using continuum mechanics, we can create more physically correct destruction simulation. This approach is in relation to destruction systems referred to as Finite Element Method (FEM is actually a technique for approximating boundary value problems, but in the context of destruction systems FEM refers a continuum mechanics-based solution, because FEM is a part of the process that makes it possible). In Finite Element Method we create a tetrahedralized version of the source mesh, as described in section 2.1.1.4. A strain, stress and stiffness matrix is used to determine the deformation and destruction effects from impact forces. [PO09] describes how this can be accomplished in a real-time game environment.

These methods, especially FEM, allow us to simulate a wide range of materials realistically; not only brittle non-deformable materials, but also materials like metal and wood. Even glass, which is a difficult special case, because of its partially liquid, partially solid nature, breaks more naturally with FEM. This makes this solution superior to the others in terms of realism.

2.1.2.4 Breakable Constraints

In the Breakable Constraints method the breakable object is simulated by using a rigid body for each fragment. The rigid bodies are connected to their neighbours with fixed constraints. The fixed constraint transfers any force or torque applied to the one rigid body it connects to the other, making the two rigid bodies act as one. These connections are then broken if an impact has a impulse over a given threshold.

Fixed constraints in rigid body physics systems are not perfectly stiff, which means that an object made out of small fragments connected by fixed constraints will bend and wobble on impact resulting in a soft body-like effect, which can be desirable in some cases.

2.1.2.5 Breakable Composite Rigid Body

The Breakable Composite Rigid Body (BCRB) method is somewhat similar to breakable constraints. In this method neighbouring fragments are also connected and those connections are then broken on collisions. At runtime we calculate impulse and propagate it through the object. If the impulse of the impact is over a certain threshold, we weaken or break the affected connections. But the BCRB approach differs because a cluster of connected fragment have only one rigid body and only when a cluster of fragments or a single fragment breaks free, then that cluster or fragment is given a rigid body.

Disconnected clusters are determined using a connected-component algorithm. It should be noted that the inertia matrix should be appropriately updated when new rigid bodies are created.

2.1.3 Dust and Debris

The methods described to this point are used to model and simulate the larger coarse fragments, but when it comes to the smaller fragments it is more practical to use other solutions, because of the high number of fragments. [TIN09] describes how to simulate and visualize small fragments and dust using particle systems. The small fragments would be represented by individual particles and dust could be represented by semi transparent particles, similar to those used when visualizing smoke. Figure 2.7 shows destruction with dust and debris from [TIN09].

2.2 Existing Solutions

The following are some commercially used destruction solutions for games:

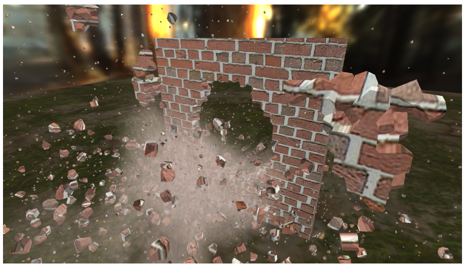
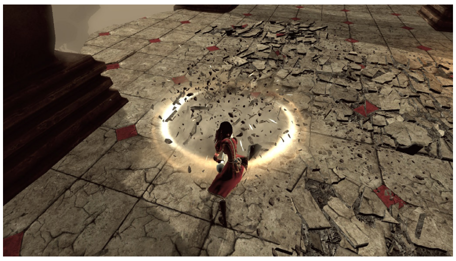


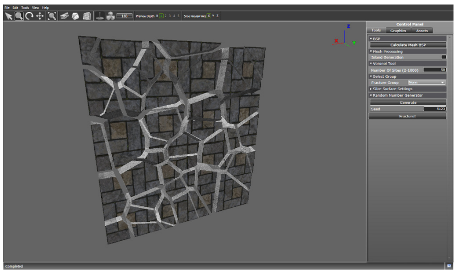
Figure 2.7: Destruction with dust and debris. Source: [TIN09]

2.2.1 PhysX Apex Destruction and Unreal Engine

Apex Destruction (Apex D) is an Apex Module for the PhysX physics engine, which can be used to create dynamic destructible object in games engine supporting the module [u0910]. Apex D is used in the Unreal Engine. Figure 2.8a shows the in-game result of using Apex D. In Apex D artists use a tool called PhysXLab to produce fragments from a source mesh using the slicing, cut-out and voronoi methods shown in figure 2.8b.



(a) Destructible stone slabs in Alice: Madness Returns.



(b) Fragments of a wall segment in PhysXLab.

Figure 2.8: Apex Destruction. Source: [u0910]

2.2.2 Geo-Mod

Geometry Modification Technology or Geo-Mod is a destruction system developed for the Red Faction games by Volition[geo]. This was accomplished using Real-time Booleans.

In the first two Red Faction games from 2001 and 2002 Geo-Mod allowed the player to alter level geometry using certain weapons. It is interesting to note that in the first game virtually anything was destructible, which introduced issues because players could blow up parts of the levels that they needed in order to progress in the game, like staircases.

2.2.3 Digital Molecular Matter

Digital Molecular Matter or DMM is a physics engine developed by Pixelux for producing realistic deformation and destruction for both movies and games using FEM as described in [PO09]. DMM is used in the game Star Wars: The Force Unleashed II as seen in figure 2.9 and the movies Avatar and Prometheus. When using DMM the developer assigns material properties to an object and the object will react realistically to impacts based on those properties.

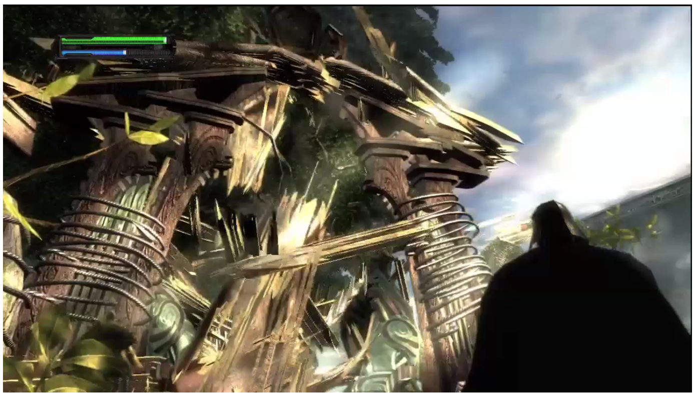


Figure 2.9: DDM Destruction in Star Wars: The Force Unleashed II. Source: [PO09]

2.3 Discussions of Methods

Manually modelling fragments (including non-real-time boolean operations) provides highest degree of creative control, but is also very costly in terms of developer time, which makes it unattractive because developers who might use this destruction system are unlikely to have an abundance of development time. It is also not an academically interesting approach. Voronoi Shattering, Slicing and convex decomposition each have various trade-offs in terms of style. Voronoi Shattering produces fragments that have a very distinct look that may be appropriate in some situations and not so in other. It also always produces convex fragments, which is positive considering that in many physics engine, including the one in Unity, only allow convex object to collide with each other. Slicing does not necessarily produce convex fragments, but convex hulls can be generated for the fragments and used for collision geometry. Slicing provides a very high degree of creative control however, especially when combined with the options of offset, angular deviation, and slicing planes distorted with noise. Convex decomposition breaks object into fragments in a very natural way, but is hardly sufficient if not combined with another method.

Regarding simulation, FEM-based destruction produces the most physically correct results, but it requires an FEM-based physics engine, which Unity does not have and building a physics engine equivalent to DMM from scratch is beyond the scope of this project. BCRB or Breakable Constraints are better solutions since they work well with rigid body physics engines. Real-time booleans is also an interesting option, because it allow us to relatively easily build an entirely destructible world, but frankly, it is not as visually impressive as a fragment-based approach. The preshatter methods all attempt to simulate how the object falls apart in some way, but Real-time booleans only allows you to punch holes in the geometry, which makes it much less realistic.

2.3.1 Experiments in Unity

To properly determine which method is best suited for implementing in Unity I produce experimental implementation and test them. Breakable Constraints and BCRB are tested, because they both work within the constraints of a rigid body physics engine, while making dynamic partial destruction system.

Unity supports fixed constraints in the form of a component called Fixed Joint. To determine if this is appropriate for use in a destruction system, I created a wall made out of cubes acting as fragments. Each cube is connected to its neighbouring cubes with fixed joints.

The earlier mentioned bending side-effect of this effect, as seen in figure 2.10, proved to be too intense and too noticeable for this method to be used in the destruction system.

Since the breakable constraints experiments proved the method to be unsatisfactory I conducted a second experiment where I created a simplified implementation of the BCRB method. In this experiment I produced a wall made

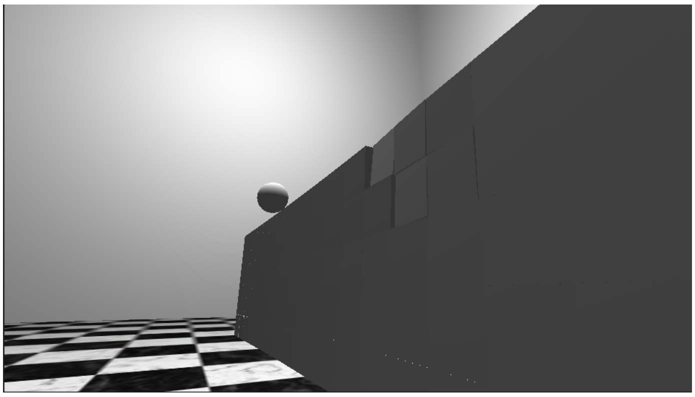


Figure 2.10: Cube Wall wobbling.

of cubes again, but this time they formed a compound collider with a single rigid body. When the wall was hit, cubes would be released from the compound collider and given their own rigid body.

With this method, there was no wobbling side effect, but the wall responded in a strange way to collisions. It would for instance fall over far too slowly, compared to a wall with the same shape and dimension using a single box collider. The reason for this behaviour is that the compound colliders in Unity are not intended to be made out of dozens, or even hundreds of primitive colliders. An example of a compound collider can be seen in figure 2.11, where a series of box colliders are combined to imitate the shape of a helicopter.

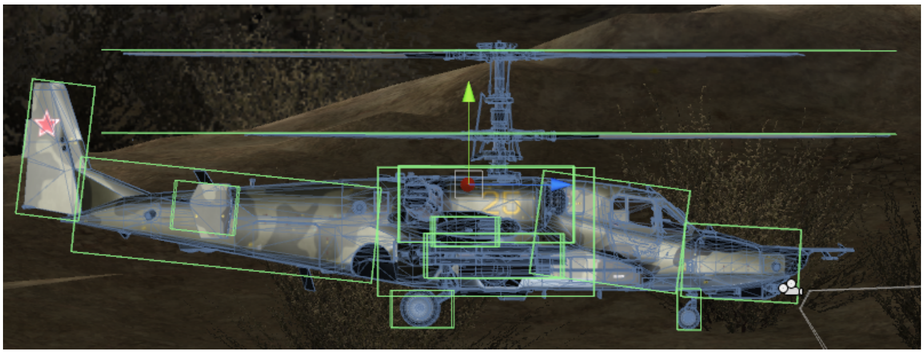


Figure 2.11: A helicopter with a compound collider set up in Unity. Source: [com]

The following chapter presents a destruction system for Unity based on the experience and lessons gained from analysing and evaluating existing solutions.

CHAPTER 3

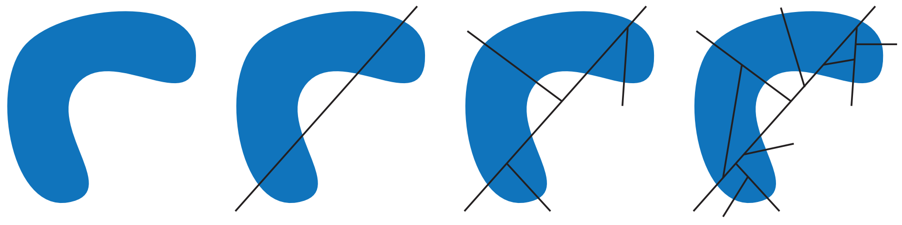
Overall Design

In this chapter, I describe the overall design of the destruction system. Based on the study of the related works and the experimental implementations, I have created the following overall design for the destruction system.

The cause of the slow behaviour discovered in the BCRB experiment was that the compound collider consisted of many concurrent collider and like many similar physics engines Unity's PhysX-based physics engine handles collisions with compound colliders by linearly iterating through the colliders that make out the compound collider. This is a design choice of the creators of the physics engine, which makes sense in terms of conventional use of compound colliders, because compound colliders are usually made out of a few primitive colliders, but in this case it causes performance to drop. To minimize the slow behaviour the amount of concurrent primitive colliders that makes out the compound collider needs to be reduced as much as possible in any given state.

3.1 Recursive Slicing and the Mesh Tree

The related works study showed that slicing is an effective way to produce fragment geometry. It potentially allows a lot of creative control and options for creating a wide range of styles of fragments. But we do not necessarily need to implement it in the same fashion as in Apex D. Instead I propose slicing the source mesh recursively, meaning that we first slice through the source mesh, then we slice through each of the resulting meshes, then we slice through each of those meshes and so forth as illustrated in figure 3.1. This is repeated until we have the desired number of fragments.



(a) Source object. **(b)** 1st iteration. **(c)** 2nd iteration. **(d)** 3rd iteration.

Figure 3.1: Recursive slicing.

This recursive slicing method is inspired by binary space partitioning, which is a way of recursively subdividing space using hyperplanes. Theses subdivisions can be organized into a BSP tree, which is a binary tree used to sort and search spatially. Each node in a BSP tree represents a hyperplane, which divides the space in two convex subspaces and each of its children represent one of the subspaces. Similarly, we create a Mesh Tree in which we store the source mesh in the root and each subdivision in a node as illustrated in figure 3.2. The leaves of the mesh tree are the fragments. Note that unlike a BSP tree, which is always binary because it subdivides convex space, the mesh tree is not necessarily binary since the shape it subdivides can be concave.

3.2 Simulation

In normal BCRB we cluster all the fragments together and under a single rigid body, which means that if you have generated 100 fragments, then you will be using 100 meshes for rendering and collision detection. However, by storing the subdivided meshes of every iteration in the recursive slicing process, we can build breakable objects with fewer concurrent meshes, by replacing certain combinations of meshes with single meshes, thus reducing the amount of concurrent colliders in the compound collider.

In a collision, where a breakable object consist of an arbitrary combination of fragments, we traverse the mesh tree and attempt to find meshes that can

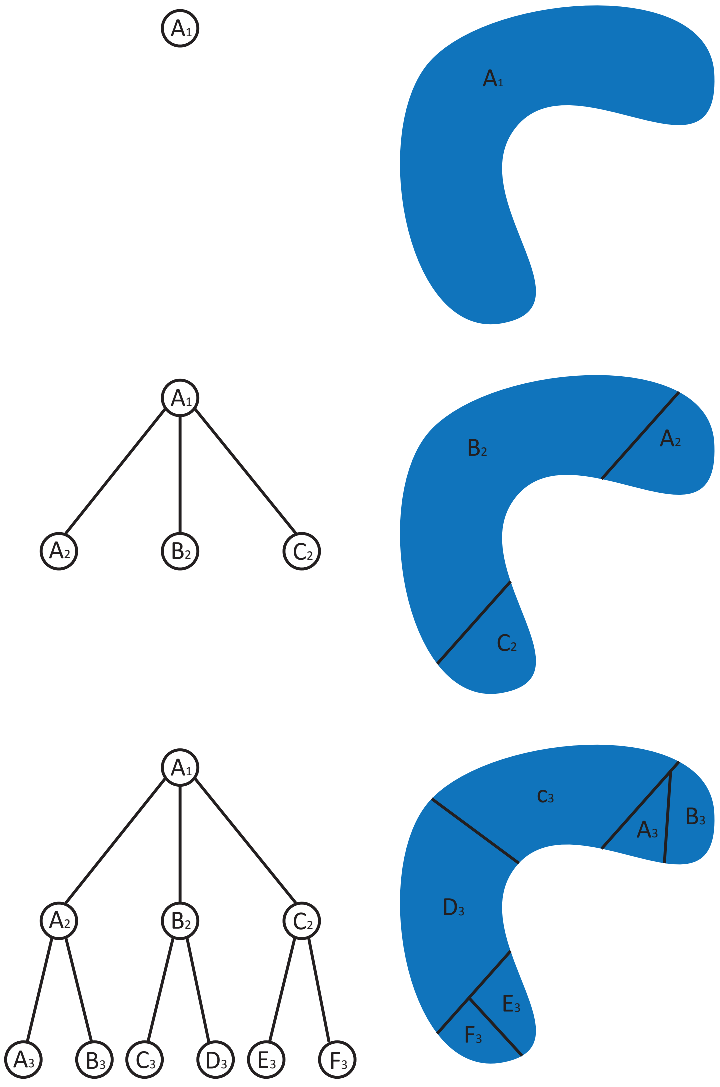


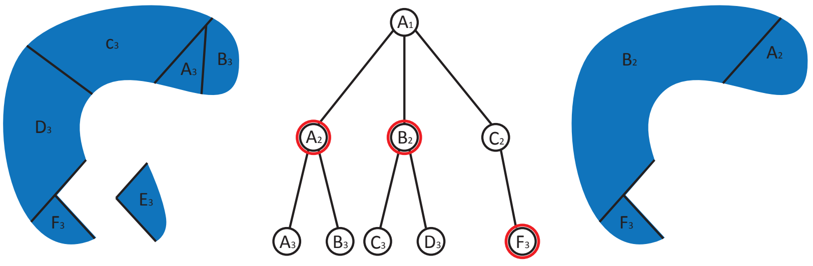
Figure 3.2: Building the mesh tree.

replace those fragment combinations and still make out the exact same shapes.

This is accomplished in the following way. Each fragment has a reference to a leaf node in the mesh tree. In a collision clusters of fragments are divided from each other and once those clusters are determined, we take the leaf nodes which the fragments clusters references to, traverse the mesh tree, and attempt to find see if the combinations of leaf meshes can be replaced by fewer meshes.

For instance, lets say that our breakable object is completely unbroken. In this case all the fragments are still included in the breakable object, therefore all the leaves are still present in the tree, hence we can use the mesh of the root node to represent the breakable object, which incidentally is the source mesh. So for an unbroken breakable object we only use the source mesh.

Let us consider a different example. In this case, the fragment E3 is broken off the object from figure 3.2 as illustrated in figure 3.3a and 3.3b where E3 is now disconnected from the fragment cluster. Since a fragment is broken off, we traverse the mesh tree to see which mesh nodes we can use to make out our remaining object. We need to use as few pieces as possible. We cannot use the the root mesh, since its subtrees no longer have all their leaves, so we will have to consider the children of the root node. The subtrees of A2 and B2 still have all their leaves so we can use the meshes of A2 and B2 and we will not continue to their children. C2 is missing a leaf in its subtree, so we have to continue to its children which only has F3 remaining. F3 is a leaf so we use it. In this example we only use 3 meshes for collision and rendering geometry instead of 5 which would have been the case with normal BCRB as seen in figure 3.3c.



(a) . **(b)** . **(c)** .

Figure 3.3: Broken object with minimal meshes from mesh tree.

The following two chapters describe how fragments are modelled and how the breakage simulation described in this chapter works in detail.

Chapter 4

Modelling Fragments

In this chapter, I describe how the geometry of the source mesh is processed to produce fragments by recursive slicing and closing the resulting gaps.

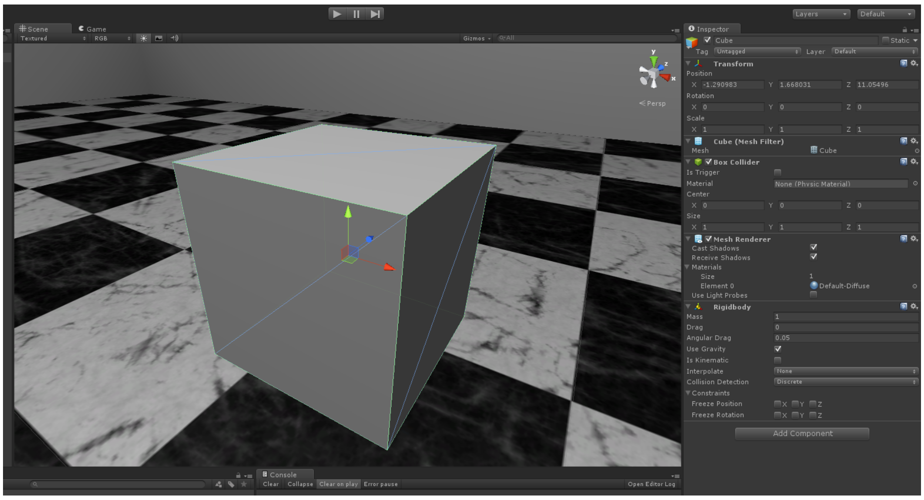
The Mesh Tree is built by first assigning the source mesh in the root of the tree, then starting a recursive process that slices the geometry of the nodes until the desired number of fragments has been reached. Mesh slicing is achieved by creating an empty meshes for each side of the plane, then sorting the triangles from source mesh based on which side of the plane they are on. The triangles intersecting the plane are split and those resulting triangle are sorted. After that the polygons faces are generated to close the two meshes on either side of the plane. Finally I check if the two meshes contain multiple hulls each, if so I create meshes for containing each hull. Listing 4.1 shows pseudo code which describes the process.

```
...
void BuildMeshTree(Mesh sourceMesh, int
   numberOfFragments)
{
    Instantiate the mesh tree and assign the source
       mesh in the root;
    SplitNode(meshTreeRoot);
}

void SplitNode(meshTreeNode)
{
    Instantiate random plane that goes through the
       center of mass of the mesh in the input mesh
       tree node;

    Mesh[] slicedResultMeshes = SliceMesh(plane,
       meshTreeNode.mesh);
    Create child node to the input node for each
       slice result mesh;

    foreach (Mesh in sliceResultMeshes)
    {
        if (number of fragments has not been reached)
        {
            SplitNode(chlidNode);
        }
    }
}

void SliceMesh(Plane, Mesh)
{
    Triangles that do not intersect the plane are
       sorted based on which side of the plane they
       are on;
    Triangles that do intersect the plane are split
       into three triangles and sorted based on which
       side of the plane they are on;
    Faces are generated to cover gaps;

    In case the slice results in more than two hulls,
       those hulls are separated so that they are
       each contained in their own mesh;

    return sliceResultMeshes;
}
...
```

Listing 4.1: Pseudo code for building the Mesh Tree

4.1 Unity Development in a Nutshell

One of the advantages of working with a commercial game engine like Unity is that it has useful tools and workflows that can simplify the development process. Unity has a component-based architecture, meaning that everything the developer works with in Unity are game objects, which by default only has a transform and the developer can add components to them to add functionality. For instance, consider a developer wanting to create a box that can be affected by physics. The developer would create an empty gameobject, add a mesh filter component, select a Cube mesh for that meshfilter, add a mesh renderer component, add a box collider component, and add a rigidbody component as seen in figure 4.1. Unity allows you to create your own components called behaviour script, which can be written in C#, Boo or JavaScript. This is how programming is done in Unity and this is how the destruction system is implemented. Unity's scripting interface contain useful classes such as a Vector class, a Transform class, a Plane class, a Mesh class and so forth.



```
Cube                                   Static
Tag  Untagged        Layer  Default
Transform
Position  X -1.290983  Y 1.668031  Z 11.05496
Rotation  X 0          Y 0         Z 0
Scale     X 1          Y 1         Z 1
Cube (Mesh Filter)
Mesh                      Cube
Box Collider
Is Trigger
Material                  None (Physic Material)
Center    X 0          Y 0         Z 0
Size      X 1          Y 1         Z 1
Mesh Renderer
Cast Shadows              ✓
Receive Shadows           ✓
Materials
  Size                    1
  Element 0               Default-Diffuse
Use Light Probes
Rigidbody
Mass                      1
Drag                      0
Angular Drag              0.05
Use Gravity               ✓
Is Kinematic
Interpolate               None
Collision Detection       Discrete
Constraints
  Freeze Position         X Y Z
  Freeze Rotation         X Y Z
```

Figure 4.1: A box in Unity.

In Unity all meshes are contained in instances of the Mesh class. The Mesh class has an integer array called `triangles` which contains indices into other arrays of various types in the Mesh class namely `vertices`, `uv`, `normals`, `tangents` and `colors`. The `triangles` array is organized 3 by 3, so that the indices of the first triangle are the first three indices in the the `triangles` array, the indices of the second triangle are the next the three indices in `triangles`, and so forth. Listing 4.2 shows a unity behaviour script that replace the mesh of a gameobject with a single triangle.

```
...
using UnityEngine;
using System.Collections;

public class example : MonoBehaviour
{
    void Start()
    {
        Mesh mesh = new Mesh();
        mesh.vertices = new Vector3[] { new
           Vector3(0f, 0f, 0f), new Vector3(1f, 0f,
           0f), new Vector3(0f, 1f, 0f) };
        mesh.uv = new Vector2[] { new Vector2(0f,
           0f), new Vector2(1f, 0f), new Vector2(0f,
           1f) };
        mesh.triangles = new int[] { 0, 1, 2 };

        GetComponent<MeshFilter>().mesh = mesh;
    }
}
...
```

Listing 4.2: A Unity script that replaces the mesh of a gameobject to a single triangle.

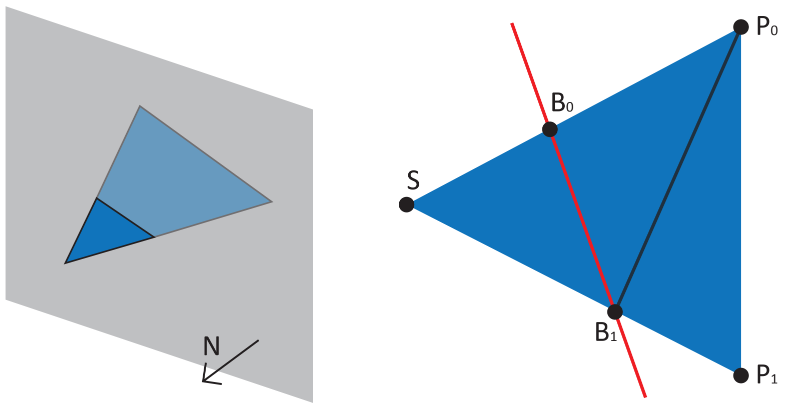
Since there is no easy way of knowing how many triangles and vertices the subdivisions are going to have after the slice, I have created a class called `MeshBuilder` that makes it possible to accumulate mesh data triangle by triangle using dynamic lists. When all the mesh data is accumulated `MeshBuilder` can output a `Mesh` object.

4.2 Sorting and Splitting Triangles

[Ebe02] describes the process of slicing a mesh a plane in mathematical detail.

The first part of the process of splitting the mesh is to sort the triangles not intersecting the plane based on which side of the plane that they are on and split the ones that do intersect the plane. First we create two `MeshBuilder` objects; one for each side of the plane. Then we iterate over all the the triangles. The triangles, where all the vertices are on one side of the plane, are accumulated in the mesh builders; if they are all on the negative side they go in the negative-side-mesh builder and if they are all on the positive side of the plane they go in positive-side-mesh builder.

Whenever we encounter a triangle where one vertice is on the one side of the plane and the other two are on the other side as illustrated in figure 4.2a, then that triangle is intersecting the plane and it needs to be spilt. This is achieved by first determining which of the three vertices are alone and which are together. The lonely vertex is denoted as S for single in figure 4.2b and the two vertices that are together are denoted as P_0 and P_1 for pair. The red line denotes the plane. To find the two border vertices we perform a raytrace from S to P_0 to find B_0 and from S to P_1 to find B_1. Normals and UVs are found through interpolation. Finally we produce new triangles as indicated in 4.2b with the same orientation as the source triangle had and pass them to their respective mesh builders. The B_0-B_1-edge is saved in a list, because we need it for triangulating the cap polygon later on.



(a) Triangle intersecting the plane. **(b)** A sliced triangle.

Figure 4.2: Slicing a triangle.

4.3 Triangulating Caps

At this point the mesh has been sliced into two new meshes which both have gaps where they were sliced as seen in figure 4.3a. To cover up these gaps we need to generate polygons, which will be referred to as caps, which cap be seen in figure 4.3b. This is achieved by first linking the border edges that was saved in the previous step together into planar straight-line graphs (PSLGs), then passing them to a triangulator, which produces triangle polygons and then finally passing those triangles to their respective mesh builders.

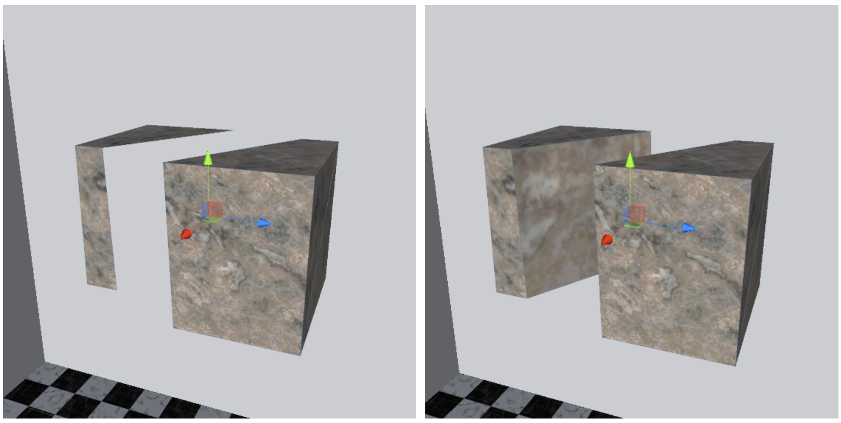

(a) A sliced cube without a cap. (b) A sliced cube with a cap.

Figure 4.3: Cap.

```
...
void TriangulateCap(BorderEdges)
{
    Initiate list of VertexLoops;

    while(List of border edges is not empty)
    {
        VertexLoops.Add(ExtractVertexLoop(BorderEdges));
    }

    Determine which vertex loops outline polygons and
       which outline holes based their orientation
       and accumulate them into a single PSLG;

    Pass the PSLG to Triangle;

    Add cap triangles on mesh;
}
...
```

Listing 4.3: Pseudo code for triangulating caps.

4.3.1 Linking Border Edges

The first part of creating triangles for the gap is to organize the border edges into a PSLG. If we are dealing with an arbitrary concave mesh then we may have to produce multiple polygons. Those polygon may have holes in them and there might even be smaller polygons within those holes. We have to anticipate any configuration of multiple polygons and holes if the fragment modeller is to work with any arbitrary mesh. However, we have to disallow meshes that are not watertight for the border edge linker to work. In conclusion border edges might contain multiple loops of vertices, which needs to be extracted one at a time.

Extracting a loop from the list of border edges is done as follows: Every edge in the border edge list connects two vertices, which will be referred to as vertex A and vertex B. An arbitrary edge (for instance the first edge) is chosen from the border edge list. The one vertex, vertex A, which the edges connects, is denoted as an endpoint and the other, vertex B, is added to a vertex loop list, then we iterate through the list of border edges until we find an edge where its vertex A matches the last vertex added to the vertex loop list. For the edge we find, we add vertex B to the vertex loop list and remove it from the border edge list. If the new vertex B matches the end point, the loop has been closed and it is returned otherwise the border edges are iterated through again. The process can be seen as pseudo code in listing 4.4.

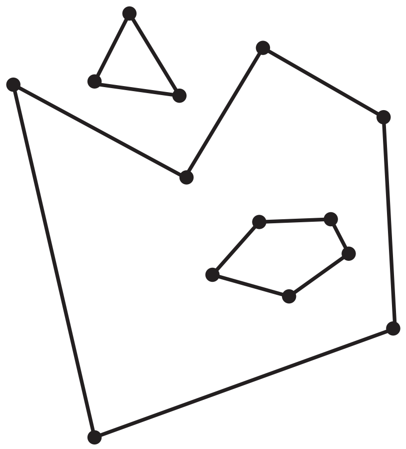


Figure 4.4: A PSLG containing a small triagular polygon and a larger more complex polygon with a hole.

```
...
VertexLoop ExtractVertexLoop(BorderEdges)
```

```
{
    Instantiate VertexLoop; // A list of vertices

    // Each edge has a A and a B vertex.
    EndPointVertex = BorderEdges[0].A;
    VertexLoop.Add(BorderEdges[0].B);
    BorderEdges.Remove(BorderEdges[0]);

    do
    {
        foreach (Edge in BorderEdges)
        {
            if (VertexLoop.LastVertex == Edge.A)
            {
                VertexLoop.Add(Edge.B);
                BorderEdges.Remove(Edge);
                if (Edge.B == EndPointVertex)
                {
                    return VertexLoop;
                }
            }
        }

    }
    while (true);
}
...
```

Listing 4.4: Pseudo code for converting border edges to vertex loops.

If there is still border edges left in the list of border edges after linking edges, then it means that there are still unlinked loops in the list of border edges and linking process is repeated, so that those border edges are also converted to vertex loops.

4.3.2 Creating the PSLG

Now that all the border edges have been linked together to form loops of vertices, we have to determine which loops outline polygons, and which outline holes in polygons and accumulate this data into a PSLG. Determining whether a loop of vertices is a polygon or hole outline is done by checking the orientation of the loop around the slicing normal of the plane. The orientation of the polygon is found by calculating the area of the polygon. For every edge in the given loop of vertices, we create a triangle consisting of that edge, and an edge from each of the two vertices to the origin in such a way that we preserve the orientation of the edge between to loop vertices. These triangles are divided into two lists based on their orientation relative to the plane normal. In figure 4.5 the orientations of the vertex loops in a PSLG can be seen. The hole has opposite orientation of the polygons.

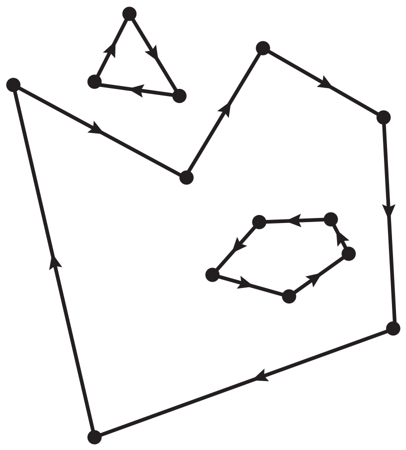


Figure 4.5: The orientation of vertex loops in the PSLG.

The area is calculated with the following formula:

$$A = \sum_{i=1}^{n-1} \left(\frac{|V_i \times V_{i+1}|}{2} \right) - \sum_{j=1}^{n-1} \left(\frac{|V_j \times V_{j+1}|}{2} \right) \tag{4.1}$$

The polygon area A is found by taking the sum of the areas of the triangles that are oriented counter clockwise around the slicing plane normal and subtracting the sum of the areas of the triangles that are oriented clockwise around the slicing plane normal. The point in calculating the polygon area in this way is that if we are on the positive side of the slicing plane, the area will be positive if the loop vertices outline a polygon and negative if they outline a hole. On the negative side of the plane, it is reversed so that the area will be negative if the loop vertices outline a polygon and negative if they outline a hole. So essentially this way we determine whether a vertex loop outlines a polygon or a hole.

4.3.3 Triangulation

Now that we know which loops outline holes and which outline polygons, they are accumulated into a single PSLG, which can be used for triangulation. Since the PSLGs that are produced in the previous step can potentially be very complex, considering that they can contain multiple polygons, holes in polygons, polygons within holes in polygons, and worse, we need a fairly advanced triangulation algorithm. Constrained Delaunay Triangulation [Che89] (CDT) triangulates with constraints given in the form of a PSLG containing vertices and line segments as seen in figure 4.6. CDT fulfills the requirements, but during the development of the fragment modeler it was assessed that implementing CDT from scratch for Unity would take too long and there did not already exist such an implementation. Instead it was decided that Triangle [She96], a 2D mesh generator and CDT-based triangulator, would be used for triangulation.

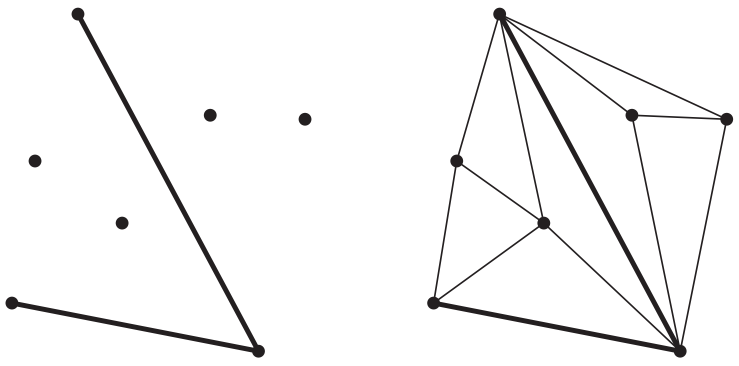

(a) An input PSLG. **(b)** CDT output.

Figure 4.6: Constrained Delaunay Triangulation.

To use Triangle I have written a Triangle interface for Unity. The Triangle interface works in the following way: First it produces a .poly file, which is the format that Triangle uses to represent PSLGs. A .poly file also needs to contain a point within each hole which Triangle uses to eat away triangles, spreading out from each hole, point until its progress is blocked by PSLG segments. Finding this point is achieved by choosing a random point within the bounds of the hole. If that point is within the hole but not within any polygons in the hole then that point is chosen, otherwise we choose a new random point and repeat the check. After the .poly has been produced, Triangle is executed with the .poly files as input. Triangle uses CDT on the PSLG and outputs a .poly containing vertices and a .ele files which contains references to the vertices in the .poly file 3-by-3 representing triangles. This data is then read from those files and returned. Pseudo code for the process can be seen in listing 4.6. The triangulated result of the PSLG illustrated in figure 4.4 can be seen in figure 4.7.

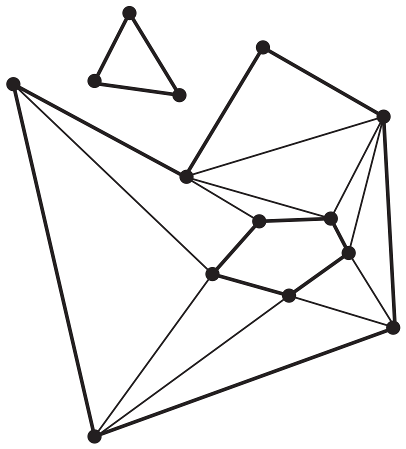


Figure 4.7: The triangulation result of the polygon.

```
...
void Triangulate(PSLG)
{
    Write a .poly containing the PSLG information;
    Execute Triangle on the .poly file;
    Read the output, an .ele and an .poly file, from
       Triangle;

    return newTriangles;
}
...
```

Listing 4.5: Pseudo code

Since the output of the triangulation is two-dimensional it is ideal for texture coordinates for the new vertices. Normals are simply perpendicular to the slice plane.

After Triangle has generated the cap triangles, they are added to the fragment mesh.

4.4 Separating Hulls

In some cases a slice may result in more than one hull in the generated meshes. In that case the mesh has to divided into more meshes so that each hull is contained within its own mesh. This is achieved by creating a list with all the unvisited triangles; initially this contains all the triangles. Then we create a list of hulls which initially just contains one random triangle from unhulled. That triangle is removed from unhulled. Then we visit all the unhulled triangles and add them to the hull if they have a vertex in common with that hull. We keep repeating that step until the number of triangle in that hull does not grow any more. Then if there are still unhulled triangles we repeat the process.

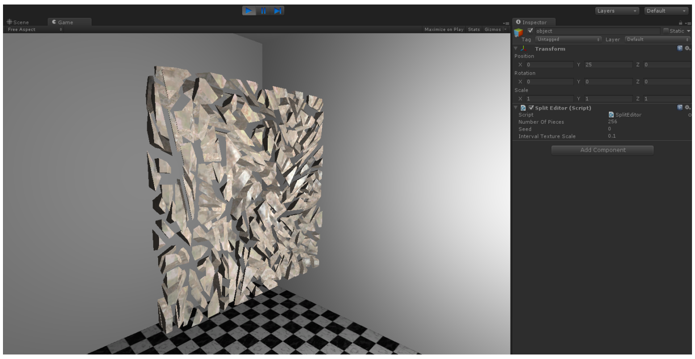
```
...
Mesh[] SeperateHulls(mesh)
{
    unhulledTriangles = All triangles from the input
       mesh;
    hulls;
    while (unhulled still contains triangles)
    {
        hull.Add(random triangle from
           unhulledTriangles);
        That triangle is removed from
           unhulledTriangles;
        do {
            foreach (triangle in unhulledTriangles)
            {
                if (triangle has a vertex in common
                   with a triangle in hull)
                {
                    hull.Add(random triangle from
                       unhulledTriangles);
                    That triangle is removed from
                       unhulledTriangles;
                }
            }
        }
        while(the number of triangles in hull has
           been increased)
        hulls.Add(hull);
    }

    return hulls;
}
...
```

Listing 4.6: Pseudo code

That concludes how the the meshes are split and how the mesh tree is generated. To preview the result of modelling fragments for a mesh the editor is used.

4.5 Editor

The editor is a tool that is intended for designers to experiment with different configurations for a mesh and preview the resulting fragments. The setting that can be configured are, number of fragments, seed for random function and internal texture scaling as seen in figure 4.8. To use the editor, the developer goes into the a special unity scene. Here there is a game object where the developer can choose a mesh and fracturing parameters, press play and preview the results.



```
object                          Static
Tag Untagged        Layer Default
Transform
Position
  X 0      Y 25     Z 0
Rotation
  X 0      Y 0      Z 0
Scale
  X 1      Y 1      Z 1
Split Editor (Script)
  Script                   SplitEditor
  Number Of Pieces         256
  Seed                     0
  Interval Texture Scale   0.1
```

Figure 4.8: Previewing a stone slab split into 256 fragments in the editor.

The following chapter describes how the Mesh Tree and fragments are used to simulate breakage.

Chapter 5

Simulation

In this chapter, I describe how the breakable objects are simulated. The purpose of the destruction simulation is to ensure that breakable objects responds to collisions in a natural way, that partial destruction is possible, that clusters of fragments can act independently if disconnected from each other, and that inertia and mass are correctly maintained for fragments, or clusters of fragments.

Simulation is achieved by using the Breakable Composite Rigid Bodies (BCRB) method, with the added complexity of using the mesh tree to reduce the concurrent number of meshes in a breakable object as described in chapter 3. This chapter elaborates on how this works.

5.1 Initial Setup

As explained in section 2.1.2.5 neighbouring fragments in BCRB are connected and those connections are broken or weakened when the breakable object collides with other objects. When those connections are broken, we use connected component analysis to determine disconnected parts for which we set up new rigid bodies. The connected parts will be referred to as BCRB Graphs, in which fragments will be represented as BCRB Nodes and the connections between them as BCRB Edges.

Ideally, only nodes with overlapping faces should be connected in the initial graph, i.e. fragments should only be connected to other fragments that they touch, but Unity does not allow manual collision detection in a preparation step. Instead, we can compare bounding boxes for each fragment mesh; if the bounding boxes of two fragment meshes overlap, we create a connection between them.

When a breakable object is initially set up the developer has a series of options that makes it possible to tweak the destruction to imitate material properties. These include, density, frailty, discrete or non-discrete connections and convex or concave colliders.

5.2 Collision

Unity allows developers to execute code at collision events in behaviour components. This is used to run the destruction simulation.

5.2.1 Damage Calculation

When a breakable object collides with another object, we calculate the impulse of that collision. If the impulse is above a certain threshold, we iterate through the edges of the graph and calculate damage based on distance between the collision contact point and the middle of the edge.

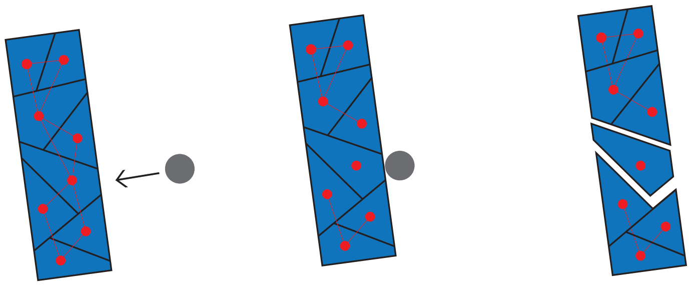
$$p = m_1 v_1 + m_2 v_2 \tag{5.1}$$

p is the impulse, m_1 is the mass of the destructible object, *v1* is the velocity of the destructible object, *m2* is the mass of the colliding object, *v2* is the velocity of the colliding object.

$$damage = f * \frac{1}{s^3} * p \tag{5.2}$$

f is the frailty factor, s distance between the collision contact point and the edge.

A result is calculated from each edge and withdrawn from its hit points. The purpose of this formula is to produce high values for edges in close proximity to the collision contact point, and smaller values as distance increases, so damage becomes very local to the collision contact point. Figure 5.1 shows how the BRCB graph is used to determine broken off pieces.



(a) Initial Setup. (b) Edges broken at collision. (c) Resulting fragment clusters.

Figure 5.1: BCRB.

There are two different options for the destruction simulation; Discrete and non-discrete. If the non-discrete option is chosen, the edges can be partially damaged. The calculated damage is subtracted from the hit points of the edge and if those hit points surpass zero the edge is broken. If the discrete option is chosen, the edges are either entirely broken or not at all; the edge can not be partially damaged. The purpose of these two options is to provide more creative control to the developer, since the two different options produce vastly different behaviour. Non-discrete connection makes it possible to accumulate damage over multiple hits. Consider for instance a coconut. Hitting it once with a hammer might not necessarily crack it, but its shell will be weakened and more likely to break when hit again. And discrete connection are used for breakable objects that do not accumulate damage, like stone. It either breaks when hit, or it does not, and is not more likely to do so when hit again.

If an edge is broken we recompute connectivity of the graph by using a Union-Find algorithm [THCS11].

```
...
BCRBGraph[] RecomputeConnectivity()
{
    For each node in the graph a list is created
        containing only that node;
    foreach (edge in originalGraph)
    {
        Merge graphs that contain the two nodes the
            edge connects;
    }

    return graphs;
}
...
```

Listing 5.1: Pseudo code for performing connected component analysis.

If the algorithm returns more than one connected component we will know that the breakable object is broken, and we set up a new breakable object for each connected component.

5.2.2 Using the Mesh Tree to Compute Minimal Meshes

If breakage occurs we need to setup a new gameobject for each connected component and we need to find the minimal number of meshes from the mesh tree. This is done by taking the graph nodes for each connected component which each have references to a mesh tree leaf and finding the meshes with the following algorithm.

```
...
Mesh[] GetMinimalMeshes(MeshTreeNode[]
    connectedComponentMeshTreeLeaves)
{
    Mesh[] meshesToBeReturned;

    Iterate through the mesh tree breadth first
    {
        if (all the leaves of the subtree of the mesh
            tree node we are currently visiting
            contains all the
            connectedComponentMeshTreeLeaves)
        {
            Add the mesh of the mesh tree node we are
                currently visiting to meshes to be
                returned;
            Do not visit the children of the node we
                are currently visiting;
        }
        else
        {
            Visit the children of the node we are
                currently visiting;
        }
    }

    return meshesToBeReturned;
}
...
```

Listing 5.2: Pseudo code for performing connected component analysis.

From the returned meshes we form a compound collider and attach it to the game object of the connected component. Note that those meshes are also used for rendering.

5.2.3 Inertia and Mass

We pass the velocity and angular velocity from the destroyed object to the rigid bodies of the resulting pieces. Mass is calculated by the physics system based on a density parameter passed as an option at setup.

5.2.4 Convex Hull Generation for Convex Colliders

Concave colliders cannot collide with each other in unity, but Unity can generate a convex hull for a concave mesh and use that for collision instead. Unfortunately this feature is buggy and will produce an error if the convex hull it generates has more than 255 triangles, whereas it should always generate a hull of 255 triangles or less. To circumvent this bug, we choose 32 random triangles from the mesh for which we want to create a convex hull, create a new mesh with those triangles and use that to generate a convex hull. Unity's convex hull generator will always produce a hull with 255 triangles or less, if the input mesh has 32 triangles. The generated hull will be a highly simplified version of the original, but it is mostly appropriate for collisions. If the convex colliders option is checked, the method is used to generate convex collider meshes for each mesh in the mesh tree.

That concludes how the destruction system works. The following chapter explores how well it actually works and what its limitations are, by exposing it to a series of test.

Chapter 6

Tests and Results

In this chapter I investigate how well the destruction system performs and what its limitations are by setting up tests and analysing the yielded results.

6.1 Test Setup

The machine I am running the tests on has the technical specifications in table 6.1.

Processor:	Dual Core AMD Opteron$_{\text{tm}}$ Processor 180 2.40 GHz
Ram:	4.00 GB
Graphics Card:	NVIDIA GeForce GTX 260
OS:	Windows 8 Pro 64-bit
Unity Version:	4.0.0f7

Table 6.1: Technical specifications of the testing machine.

6.2 Testing Fragment Generation

These tests determine various performance aspects of the fragment modeller.

6.2.1 Speed

The purpose of the test is to determine how long it takes for the fragment modeller to produce a given number of fragments. Speed is not essential for the destruction system to work, but it may influence the workflow of the developer when using the editor to experiment with the settings of the fragment modeller, because the process has to start over every time a parameter is changed. The test is executed on a cube and the results can be seen in figure 6.1.

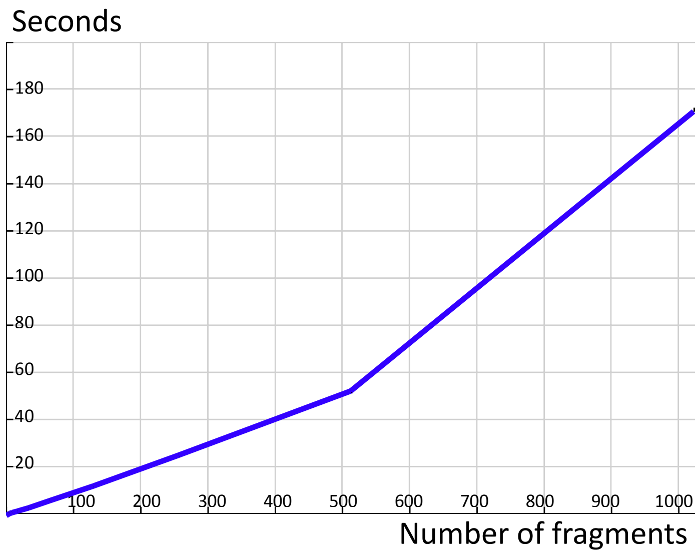
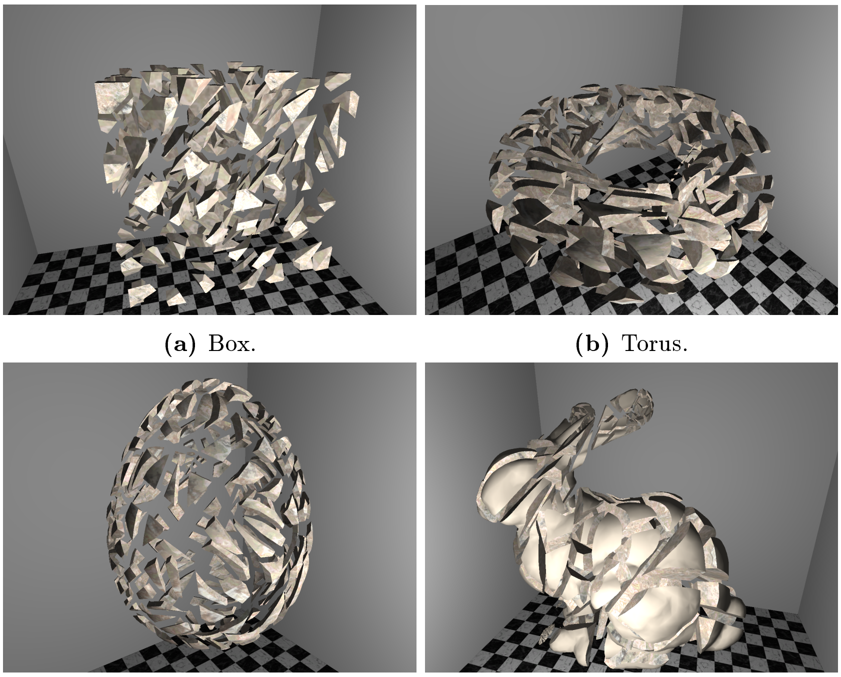


Figure 6.1: Time measurement of generating fragments compared to number of fragments.

6.2.2 Meshes

In this test the fragments modeller is used on different meshes of various geometric complexity namely a cube (24 verts, 12 tris), a torus (441 verts, 800 tris), a hollow egg shell (878 verts, 1520 tris), and a Stanford Bunny (2545 verts, 5002 tris). In all cases the mesh is split into 256 fragments.



(a) Box. **(b)** Torus.

(c) Egg Shell. **(d)** Stanford Bunny.

Figure 6.2: Various concave meshes slice into 256 fragments.

In all cases the fragment modeller successfully produces fragments. Though there is a noticeable issue, for meshes that have protrusions such as the bunny, the fragment modeller has a tendency to produce clusters of very small fragments, instead creating more even sized fragments.

6.2.3 Possible Number of Fragments

The purpose of this test is to determine how far the fragment modeller can be pushed in terms of maximum number fragments. By incrementing the number of fragments by the power of two, the maximum number of fragments can be approximated.

The fragment modeller was executed on a cube and started producing errors at 2048 fragments. It the edge linker in the cap triangulator that stops working at this point, because Unity has a tendency to produce floating point errors and when the fragments get too small Unity cannot properly compare vectors, due to these errors. This makes linking the border edges impossible.

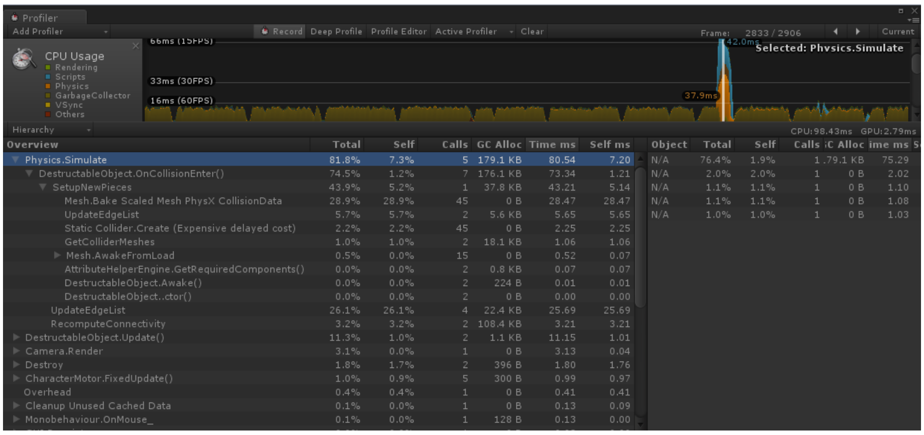
6.3 Testing Runtime Simulation

These tests determine the performance of the breakage simulation.

6.3.1 Possible Number of Fragments

In this test, the number of fragments is increased by the power of two until the lags at collisions become noticeable. The test was executed on a cube.

The lags started becoming slightly noticeable at 64 fragments and the result can be seen in figure 6.3 in Unity's Profiler where the a collision event is marked.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Physics.Simulate	81.8%	7.3%	5	179.1 KB	80.54	7.20
DestructableObject.OnCollisionEnter()	74.5%	1.2%	7	176.1 KB	73.34	1.21
SetupNewPieces	43.9%	5.2%	1	37.8 KB	43.21	5.14
Mesh.Bake Scaled Mesh PhysX CollisionData	28.9%	28.9%	45	0 B	28.47	28.47
UpdateEdgeList	5.7%	5.7%	2	5.6 KB	5.65	5.65
Static Collider.Create (Expensive delayed cost)	2.2%	2.2%	45	0 B	2.25	2.25
GetColliderMeshes	1.0%	1.0%	2	18.1 KB	1.06	1.06
Mesh.AwakeFromLoad	0.5%	0.0%	15	0 B	0.52	0.07
AttributeHelperEngine.GetRequiredComponents()	0.0%	0.0%	2	0.8 KB	0.07	0.07
DestructableObject.Awake()	0.0%	0.0%	2	224 B	0.01	0.01
DestructableObject..ctor()	0.0%	0.0%	2	0 B	0.00	0.00
UpdateEdgeList	26.1%	26.1%	4	22.4 KB	25.69	25.69
RecomputeConnectivity	3.2%	3.2%	2	108.4 KB	3.21	3.21
DestructableObject.Update()	11.3%	1.0%	2	1.1 KB	11.15	1.01
Camera.Render	3.1%	0.0%	1	0 B	3.13	0.04
Destroy	1.8%	1.7%	2	396 B	1.80	1.76
CharacterMotor.FixedUpdate()	1.0%	0.9%	5	300 B	0.99	0.97
Overhead	0.4%	0.4%	1	0 B	0.41	0.41
Cleanup Unused Cached Data	0.1%	0.0%	1	0 B	0.13	0.09
Monobehaviour.OnMouse_	0.1%	0.0%	1	128 B	0.13	0.00

Object	Total	Self	Calls	:C Alloc	ime ms
N/A	76.4%	1.9%	1	.79.1 KB	75.29
N/A	2.0%	2.0%	1	0 B	2.02
N/A	1.1%	1.1%	1	0 B	1.10
N/A	1.1%	1.1%	1	0 B	1.08
N/A	1.0%	1.0%	1	0 B	1.03

Figure 6.3: Profiler at collision.

The profiler shows that the main lag is mainly caused by two things: Setting up the new colliders and rebuilding edges when determining connected parts.

The final result of the destruction system in use can be seen in figure 6.4 which shows a wall segment and an eggshell after being partially destroyed. In both cases the objects consist of 32 fragments. In these cases the destruction is fully functional.

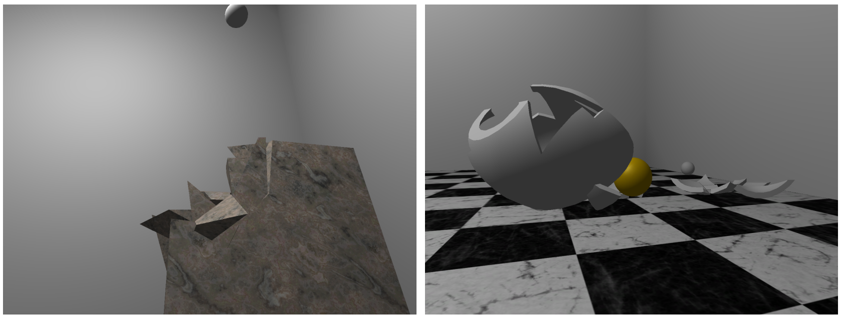

(a) A partially destroyed stone wall. **(b)** A partially destroyed eggshell.

Figure 6.4: The destruction system in action.

The following final chapter reflect on the accomplishments of this project and on the results derived from this chapter. It also describes how the destruction system can be further improved.

Chapter 7

Conclusions

In this chapter, I reflect on the process and the results and speculate on what could been done differently, and how it can be improved if given more development time.

This thesis started with a proposal for making dynamic destruction a more accessible feature to game developers making low-budget games in a fast and easy way. In doing so the potential production value of these low-budget games could be raised. By implementing a destruction system in the Unity game engine, which is highly used among low-budget game developers, the target audience could be reached. Before I started developing the destruction system, I conducted a study of related works to determine what kinds of solutions already existed. I discovered how they worked I attempted to determine if I could build my solution on the work of these other solutions. In the study of the related works it was found that a great deal of solutions already existed and that it was both possible and appropriate to reuse some of that technology to accomplish the goal of this thesis. Based on the research and experimental implementation of some of the existing methods, it was concluded that Breakable Composite Rigid Bodies (BCRB) was the best suited simulation method when combined with generating fragment geometry by recursively slicing the source mesh. BCRB worked well within the constraint of Unity's rigid body physics engine and made dynamic and partial destruction possible. The purpose of slicing the source mesh recursively was to build a Mesh Tree, which made it possible to reduce the number of concurrent meshes that represented the breakable object, thus reducing physics engine performance issues. A fragment modeller was implemented to recursively slice an object to produce fragments by dividing meshes along planes, and generating polygon faces to close the sliced meshes. BCRB was implemented to simulate how the fragments would fall apart and the mesh tree was used to reduce the concurrent number of meshes that made out the breakable object. Once the destruction system was completed, various formal tests were executed to determine how well the solution worked and what its limits were. Most significantly it was found that on the test platform, the destruction worked seamlessly until with 32 or less fragments per breakable object. At 64 fragments the system produced noticable performance spikes at collisions, which were mainly caused by having to set up the colliders and by having to rebuild the edges in a graph used in the BCRB simulation.

7.1 Discussion

The goal of this thesis was to create a destruction system that would be easy for developers to use, and that would work within the constraint of Unity. That goal has been reached. Game developers can use this system to create destructible objects within their games by taking object that they want to be destructible into an editor an experiment with the parameters to get the desired result, then add a component to the object in the game with those same parameters.

The most significant hindsight in this project has been that slicing a mesh was more troublesome than anticipated. The biggest problem when developing the mesh slicer, was closing the mesh after the were sliced, because none of the existing triangulator made for Unity were able to triangulate the complex polygons needed. Another major problem in the development process was that Unity had a tendency to produce floating point errors when dealing with vectors and Unity's vector comparison appeared to be relatively impressive. These problems made it apparent that Unity is not well suited for complex geometry processing. So in hindsight the editor should have been developed independently of Unity and results should have been exported in a way that could then imported into Unity.

7.2 Further Development

If given more development time the following implementations could improve the destruction system.

- Having to rebuild BCRB graph edges causes about half of the performance drop on breakage. This could be solved by preserving edges when determining connected components.
- The convex hull generation that was used in this project can potentially produce convex hulls that do correspond well to the source mesh, because the triangles used for it are chosen at random. A better way of creating an approximated convex hull could be to generate a discrete oriented polytope (k-DOP).
- Generating the Mesh Tree for a single object can easily take multiple minutes. It the current state of the destruction systems, this has to be done every time the game loads, which is unfortunate. This could however be circumvented by having the editor serialize the mesh tree for a given set of parameters. The mesh tree could then be loaded when the game loads, thus greatly reducing game loading time.
- Add particle systems for dust and debris.
- Make it possible to make alternate internal texturing.
- Preserve tangents. The slicer only preserves texture coordinates and normals in its current state.
- Better initial connections generation, so only truly neighbouring fragments are connected by fragments.
- Saving references to vertices instead of the vertices themselves makes it possible to avoid comparing vertices.

Ultimately if this further development was conducted, the quality of the destruction system could be raised to a point where it could be submitted to the Unity's asset distribution system, thus making it accessible to a great mass of game developers.

Appendix A

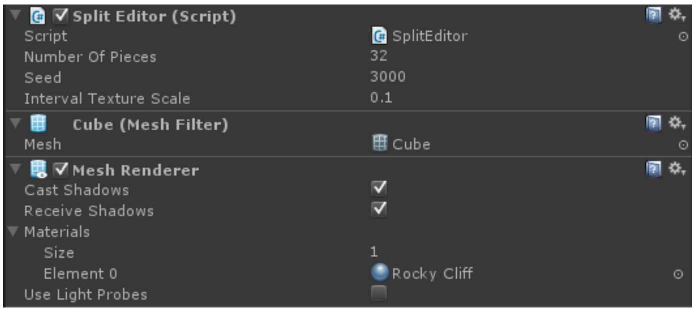
User Manual

This appendix describes how to use the destruction system in Unity.

A.1 Editor

To preview how the fragments for a given object is going to look given a certain set of parameters, the developer must do the following: Open the Editor scene, which contains a lit room, a camera with Maya-like controls and the preview object. The preview object has a mesh filter, a mesh renderer and a spilteditor component attached. The developer chooses the mesh he/she wants to preview by changing the mesh in the mesh filter. The spilteditor has several parameters as seen in figure A.1, which the developer can experiment with to find the desired fragments composition.

To preview the effects the developer must press the play button, and the fragments will be procedurally modelled. Once it is done modelling the fragments, the developer can rotate the camera around the object by holding the ALT-key, left mouse button and move the mouse. To zoom the developer holds the ALT-key, right mouse button and moves the mouse left and right. To take the fragments apart the developer holds the SHIFT-key, left mouse button and moves the mouse left and right.

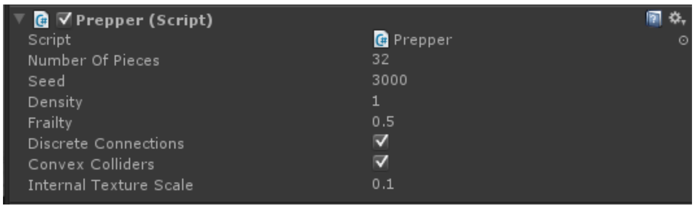


Split Editor (Script)	
Script	SplitEditor
Number Of Pieces	32
Seed	3000
Interval Texture Scale	0.1
Cube (Mesh Filter)	
Mesh	Cube
Mesh Renderer	
Cast Shadows	✓
Receive Shadows	✓
Materials	
Size	1
Element 0	Rocky Cliff
Use Light Probes	☐

Figure A.1: The editor component.

A.2 Prepper

Once the developer settles on a certain set of parameters, he/she goes to a game scene and attaches the prepper component to the object that is to be made breakable. The parameters of the prepper can be seen in figure A.2. The developer copies the parameters from the editor to the prepper and must also choose frailty and density. The prepper prepares the object when the game is loaded to make it destructible.



Prepper (Script)	
Script	Prepper
Number Of Pieces	32
Seed	3000
Density	1
Frailty	0.5
Discrete Connections	✓
Convex Colliders	✓
Internal Texture Scale	0.1

Figure A.2: The prepper component.

Once these steps have been performed the object is ready to be destroyed in the game.

Bibliography

[BNT90] John Amanatides Bruce Naylor and William Thibaultf. Merging bsp trees yields polyhedral set operations. In *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 115–124, August 1990.

[Che89] L. Paul Chew. Constrained delaunay triangulations. volume 4, pages 97–108, 1989.

[com] Creating gameobjects with multiple colliders [image heavy]. http://forum.unity3d.com/threads/123811-Creating-gameobjects-with-multiple-colliders-image-heavy.

[Cou11] Erwin Coumans. Overview of destruction and dynamic methods. Presented at Siggraph 2011, 2011.

[Cou12] Erwin Coumans. Destruction. http://www.altdevblogaday.com/2011/09/02/destructio
September 2012.

[Ebe02] David Eberly. Clipping a mesh against a plane, February 2002.

[Est11] Jose Miguel Esteve. Voronoi shattering. http://www.joesfer.com/?p=60, March 2011.

[geo] Geo-mod. http://redfaction.wikia.com/wiki/Geo-Mod.

[Led07] Hugo Ledoux. Computing the 3d voronoi diagram robustly: An easy explanation. In *ISVD '07 Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering*, pages 117–129, April 2007.

[MG09] Khaled Mamou and Faouzi Ghorbel. A simple and efficient approach for 3d mesh approximate convex decomposition. In *Image Processing (ICIP), 2009 16th IEEE International Conference*, pages 3501–3504, November 2009.

[PO09] Eric G. Parker and James F. O'Brien. Real-time deformation and fracture in a game environment. In *SCA '09 Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 165–175, 2009.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. volume 1148, pages 203–222, 1996.

[THCS11] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2011.

[TIN09] Henry Johan Takashi Imagire and Tomoyuki Nishita. A fast method for simulating destruction and the generated dust and debris. In *The Visual Computer: International Journal of Computer Graphics*, volume 25, pages 719–727, April 2009.

[u0910] Apex destruction. http://physxinfo.com/wiki/APEX_Destruction, August 2010.