Abstract

Destructible environments are a key feature to increase a player's immersion in modern games. In many games destructible environments are achieved using pre-determined destruction and animations to simulate the effect of destruction. The purpose of this project is to create a program that can generate real time destructible environments. To achieve the desired effect a connected mesh of particles will be created and the links between the individual particles will be destroyed using a combination of ray casting and the barycentric coordinates of triangles. This will allow the mesh to be split into smaller parts simulating a realistic destructible environment.

Keywords: Destruction, Splitting, Ray Casting, Collision Detection

1 Introduction

In many modern video games destructible environments are used to create a more realistic game world so that the player will achieve a sense of immersion when playing. This is usually achieved by having multiple models for each object to show various stages of destruction. Although this may be a good solution it can often create an unrealistic simulation of how the object would be destroyed under certain circumstances causing the player to have a lesser experience within the game world. In many games items in the world such as barrels or crates can be destroyed, however by using a predetermined mesh to simulate the destruction the object is destroyed in a way that is unrealistic and often repeated for every instance of that object in the game (Figure 1).

In this paper we attempt to create a system in which the objects will be destroyed in real time in order to simulate how an object would be destroyed in the real world. This effect will be achieved by creating an interconnected mesh of particles and triangles. First, a ray will be cast onto the mesh and if a collision is then detected the nearest constraint will be calculated and then destroyed as well as the triangle in which the collision occurred and any other triangle the constraint is a part of. If any constraint remains that is not a part of a triangle it will also be removed. For this solution to create truly realistic destruction the mesh required would have to be large which would create bottlenecks in the current system resulting in a dramatic decrease in overall performance due to the method used to check constraints that has been implemented.

Figure 1: Crate in Half Life 2 - A Crate in Half Life 2 being destroyed

The remainder of the report is structured as follows. In Section 2 we discuss the background of this project as well as related work completed by others. Section 3 gives an overview of the whole project, how the software is designed and the logic flow of the project. Section 4 discusses the results of the simulation and provides a description of how the simulation was implemented. Section 5 provides an evaluation of the final simulation, comparing the completed project with the original concept. The overall quality of the simulation and possible improvements will also be discussed in section 5. Finally, a summary of the project will be provided in Section 6 as well as further work that could be implemented into the project in order to improve the capabilities of the simulation.
2 Related Work

Destruction in games has been a widely researched area, with many papers being written on the different techniques used to create realistic and efficient methods to implement destruction. [Parker et al. 2009] Gives an example of how separate components can be used simultaneously in order to create a game environment in which the player can destroy objects in real time with a realistic manner. It also shows how game engineers can work with artists on order to create the destruction wanted. [Teschner et al. 2003] Attempt to create a new approach to allow for collisions and self-collisions for dynamically deforming objects by the use of spatial hashing. If this technique were to be implemented into our simulation it would allow for a much more realistic looking simulation as the self-collision could stop any problems that occur when the cube passes through itself as well as decrease the time needed to run the simulation. [Steinemann et al. 2006] Create an efficient method of splitting deformable objects in non-determined ways. The neighbour and dynamic neighbourhood updates provided in the paper gives a good technique that could be implemented to help optimise the simulation when splitting the objects in real time. It could also be used in the creation of meshes of different shape and size that were to be destroyed using our method. The inclusion of AABB trees would also provide a method of collision detection for the simulation when sections of the cube are split from the main cube. [van den Bergen et al. 1998] Provides a good method for detecting collision between objects that are of a complex shape or size. Van Den Bergen creates a way to increase the performance of AABB trees and speed up the overlap test to allow for faster collision detection. A method for updating the AABB tree when the model is deformed is also shown to allow for faster deformation updates that OOBs.

Splitting cubes has been achieved before, [Pietroni et al.] created a method in which the cube can be split in ways that are not pre-determined and can be split along multiple points. This was achieved by putting the object that is to be deformed into a regular grid in order to find the points in the objects that were to be split. In their method, when a cube is being split, the nodes in the split are determined and changes the shape of the cells in the regular grid. The shape is then deformed to create a realistic cutting algorithm. [Otaduy et al.] Creates a hierarchal system to handle the multiple collision that occur when fracturing an object. This is achieved by storing the bounding volumes of each fracture in a hierarchy using trees. The system that is achieved allows for dynamic fracturing with relative speed. This would be implanted into our system in order to increase the optimisation of the simulation.

3 Methodology

When the simulation starts a cube is spawned in the simulation. The user can then spawn in a cube of another size, apply gravity to the cube or click onto the cube. When the User clicks on the cube a ray is cast from the screen onto the cube. The collision checking function is then called and if the ray does not intersect with a triangle then nothing happens. If however, a collision is detected then the simulation finds out where the intersection occurs. Since the ray can intersect multiple triangles at once the simulation finds the triangle that causes the ray to have the shortest length, this is the triangle that is selected to be destroyed as it was the one closest to the screen when the ray was cast. (Figure 5)

The triangle is then passed to the function which calculates the nearest constraint to the intersection point. To do this, the area of the triangle created between the intersection point and two of the points on the triangle is calculated (Equation 1). This is done for each combination of points giving three different values. The points that give the smallest area is then used as the constraint that is to be destroyed. (Figure 4)

Area of Triangle Made By Two Vectors (1) below:

\[ Area = \sqrt{\|A \times B\|} \]  

(1)

Where A and B are two vectors made from the intersection point and a point of the triangle.

The two points are passed into the remove constraint function which loops through all the constraints to check if the two points make up the constraint. If they do then the constraint is removed.
Once the remove constraint function is completed, the two points are then passed to the remove triangle function. This function works in a similar way to the remove constraint function except it has to check all three points of the triangle instead of just the two points it must check for the line. (Image 7)

With both of these methods complete the simulation must then check if there are any constraints left that are not a part of a triangle. The method achieves this by checking each individual constraint against every triangle. If the constraint is a part of an existing triangle then the constraint is left alone. If in fact the constraint is not a part of a triangle then it is removed (Figure 8). Once all the checks have been completed the cube is redrawn with the appropriate sections removed. An overview of the whole process is given in Figure 2, with the dependencies shown in Figure 3.

The control mechanisms work as follows:

- Right Mouse Button: Move the camera around the simulation.
- Right Mouse Button + Left Control: Zoom in and out of the screen.
- Left Mouse Button: Pressing the left mouse button will allow the user to cast a single ray onto the cube.
- Left Mouse Button + Left Control: Holding down the left mouse button and the left control button will allow the user to cast a multiple rays onto the cube.
- Mouse Wheel: Holding down the mouse wheel will allow the user to move the cameras position in the simulation.
- Arrow Keys: Move the cube around the bounding box.
• F1,F2,F3,F4: This will allow the user to spawn cubes of different size.
• F6: This will apply gravity to the cube.

4 Evaluation

The Simulation created in this report has some changes from the original concept proposed for the project. One of the main changes from the original design was not destroying the individual particle but instead destroying the nearest link and removing any triangle that the link was a part of, however this was purely a stylistic choice as it created a better looking effect once it was implemented within the simulation. Also, reconnecting particles was removed due to time constraints.

The Simulation runs in real time when destroying the links of small cubes but suffers from a large drop in frame rate as the number of links increases. This is due to the algorithm that checks if any constraints are not a part of a triangle. To find extra constraints the method gets the first constraint in the list and check it against each triangle in the list. When it is found to be a part of a triangle then the program exits the loop and moves onto the next constraint in the list. For small cubes this does not cause a problem as there are relatively small numbers of constraints and triangles. If however, a large cube is used the number of times it would have to loop through each constraint and triangle grows exponentially in size. For example a 3x3x3 has 27 particles, 90 constraints and 96 triangles. This would result in 3644 checks within the method. If given a 10x10x10 cube it would consists of 1000 particles, 5130 constraints and 8748 triangles. This would mean the method must

would make a total of 21,303,342 checks to make sure there are no constraints that are not a part of a triangle. (Figure 9)

This could be improved by either checking only the links that were a part of the destroyed triangle or dividing the cube into sections and checking the constraints on the section in which a triangle was removed. The use of multi-threading would also greatly increase the overall quality of the simulation as it would allow a large increase in the total number of particles and constraints as well as decrease the time it takes to perform the collision checks and the method used to remove constraints and triangles resulting in an increase in total performance.

When the cube is split into multiple parts the separate sections pass
through each other due to a lack of cube to cube collision detection. The Simulation would be greatly improved by the addition of collisions between parts of the cube that have been split from the original cube and could be achieved by generating an Axis Angle Bounding Box or an Orientated Bounding Box for each new section when it is split from the original cube. The lack of inter-cube collision detection also results in the triangles being able to pass through each other when connected to the same cube. Again, this would be resolved by implementing a method to ensure that the triangles collide with each other and not allowing the triangles to intersect.

5 Summary

In conclusion, the finished project produced an effective method of destroying objects in real time and fulfils the main goal of the original design. However, some features such as cube to cube collision were not implemented due to time constraints. The simulation runs smoothly with relatively small cubes but further optimisation will be required if larger meshes were to be implemented.

Given more time, there are many features that could be implemented into the simulation. The main feature that would be added would be collision detection between sections of the cube that have been split from the original. Another feature that could be implemented would be to add values for stress and strain between particles and remove the constraints if the force between the two particles is too high. Having a method to remove multiple constraints would also be a feature that could be implemented into the simulation as well as a technique in which a force could be spread across multiple constraints and if the force is too high the constraint is removed.

References

Otaduy, M. A., Chassot, O., Steinemann, D., and Gross, M. Balanced hierarchies for collision detection between fracturing objects. 2


Pietroni, N., Gionovelli, F., Cignoni, P., and Scopigno, R. Splitting cubes: a fast and robust technique for virtual cutting. 2

