Achieving Realtime Destruction

Matt Filer
17021871

University of the West of England

May 7, 2020

This report outlines the creation of a mesh destruction system in Unity, and the background research undertaken to implement it.

1 Introduction

Realtime destruction has always been a desirable feature for games, with simple destruction being implemented as far back as Asteroids in 1979. As fidelity has increased over time however, realistic and dynamic on the fly destruction has become a largely unachievable task.

In practice, various methods are utilised to fake destruction: typically this involves artists producing destroyed asset segments which spawn to replace the original, hidden with some kind of particle system. While this isn't a bad solution as it allows strong art direction, it limits the number of possible fractures and limits results to appear visually the same every time.

This report aims to explore past research into the field, and details an implementation of an alternate realtime mesh destruction system in Unity.

2 Related Work

There are various famous examples of the asset swap technique, such as "leveloution" in Battlefield 4. While publicised mainly for its large scale destruction system which was calculated offline and played at runtime as an animation sequence, a smaller micro destruction system was also present. This micro system involved certain assets being split into segments with individual damage states, changed over time based on damage dealt by the player. Asset swaps between states were hidden by particle effects, such as dust (Harrison, 2014).

Cloud computing was a solution touted by Microsoft in the lead-up to the release of Crackdown 3 for large-scale realistic destruction. In principle, calculating mesh destruction remotely would take strain off of clients and allow for realtime destruction to occur on a never before seen scale. While this did make the shipped version of Crackdown 3, it had to be heavily reduced in complexity from the initial E3 demo to allow for the number of lobby instances due to the compute demand placed on the servers (Leadbetter, 2019).

In an older example presented at SIGGRAPH 2009, a complex destruction system was developed for Star Wars: The Force Unleased which focussed on producing realtime fractures as well as more generic destruction. This system was optimised to the point that it was able to be computed locally on PlayStation 3 and Xbox 360, with impressive small-scale results. While the system ran in realtime utilising a corotational tetrahedral finite element method, it was also able to be influenced by artists in order to achieve destruction that didn't interfere with level design, without designers requiring technical knowledge (Parker and O'Brien, 2009).

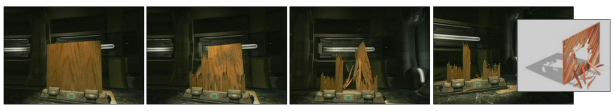

Figure 1: *A demonstration of the system developed by Parker and O'Brien, 2009.*

An alternate method was proposed in 2013 through NVIDIA which involved applying user-defined fracture patterns to meshes dependent on physics impact location using volumetric approximate convex decompositions. This system works by creating compound meshes from a given visual mesh using Voronoi nodes placed by an artist. These volumetrically created convexes can then have the fracture pattern applied to them, making it cheaper to compute the results to out-

put back as a visual mesh. This system performs well, with a given arena destruction example simulating below 50ms, however the system requires quite a lot of manual input with setting up the Voronoi nodes and fracture patterns (Müller, Chentanez, and Kim, 2013). A similar project was attempted in 2017 without the volumetric component, and as expected, performed far worse at runtime (Grönberg, 2017).

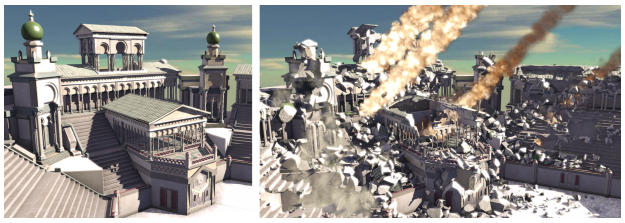

Figure 2: *A demonstration of the system developed by Müller, Chentanez, and Kim, 2013.*

Recently, large-scale destruction technology has been implemented into public engines, such as Unreal Engine 4. This implementation, known as Chaos, allows for mesh destruction at runtime through multiple methods. The key method utilised by Chaos is still similar to older methods of pre-defining breaks in the model, being pre-computed offline, then splitting the mesh into the offline generated segments at runtime when damaged. The system also provides some other methods however, such as voxelising the scene and computing signed distance fields throughout, to be able to dynamically split geometry – this can however have performance impacts, particularly on less modern systems (*Chaos Destruction*).

3 Method

For this project's implementation in Unity, a fully automated threaded recursive algorithm was developed. This implementation is proposed as an alternative to methods found in examples under *Related Work*, and is remotely similar to some ideas presented by Lien and Amato, 2006.

The method detailed below describes a single iteration of the recursive fracturing process which loops a dynamic number of times based on the object's impact velocity, modulated by its material's strength value. The algorithm is only called if the object's impact velocity exceeds a threshold as defined by its material's strength and required breaking force. Material properties are configured through a toolkit included with the project.

To split the mesh of the current GameObject, a plane is created from a given point on its surface across a randomised direction. This point is taken from impact location, and direction could also take this into account. Using the generated plane as the cutting line, each triangle within the mesh is evaluated against the cut to see if it lies on the left or right side. Triangles are individually added to two new left and right mesh objects respectively once tested, depending on the side they are judged to be on.

If a triangle fails to lie on the left or right, it must lie across the cutting plane, and so is broken up into four new triangles. To calculate these new triangles, the cutting plane is re-used to evaluate the original triangle's three vertices individually, rather than considering it as a whole. The side of the cut that gains two vertices is turned into three triangles, and the side that gains one vertex is turned into one triangle. To calculate these triangles, the point of intersection of the cut along the original triangle's edges is taken, along with the mid-point of the edge between the two vertices on one side. This mid-point forms a vertex for the new three triangles, and the intersections form the remaining vertexes for these. On the other side, the intersections and original single vertex form the other triangle. The four new triangles are added to their respective sides.

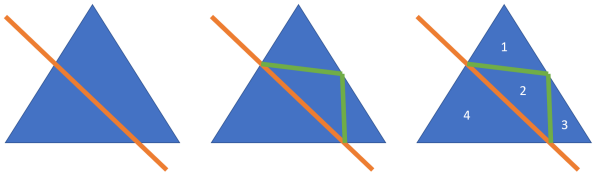

Figure 3: *A visual example of the triangle splitting across the cut line.*

There are now two individual meshes created from the original mesh, however where the cut has happened there will be a gap in the mesh through the new area that is visible. To solve this, the mid-point of all of the newly exposed vertices is calculated, and this allows the creation of triangles from two newly exposed vertices to the centre of the exposed area. Creating triangles around the newly exposed region using two exposed vertices at a time will eventually cover the exposed area with new faces; which can be added to the new meshes. We now have two fully cut meshes from the original mesh, with no unwanted holes.

Although we have all of our vertex positions calculated, UV coordinates must also be calculated to support textured meshes. This is done very simply on the newly produced cut surface triangles by subtracting the distance of the cut down the original triangle from the original UV difference. For triangles produced to fill the exposed hole in the new meshes, a simple colour check is performed on the original material, and this is applied to the newly produced triangles; as the centre of the mesh wouldn't match a texture image.

This whole process happens recursively for the number of iterations requested by the manager system, which varies depending on the object's material strength and required initial impact force. Each iteration of the recursion runs on its own thread (where possible) in order to attempt to reduce frame stutter when being calculated.

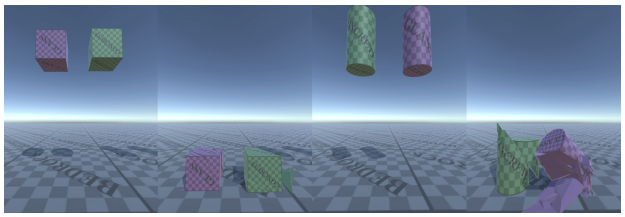

Figure 4: *A demo of dropping cubes and cylinders with this system.*

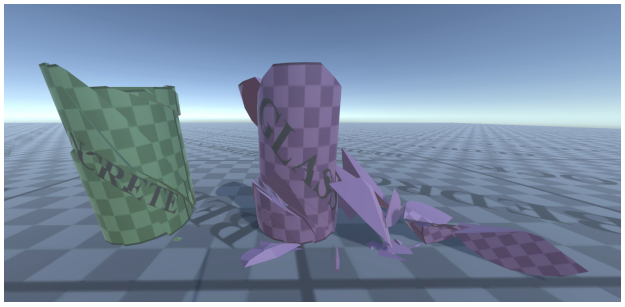

Figure 5: *A comparison of dropping concrete and glass materials from the same height with this system.*

4 Evaluation

While this implementation works, it can be inefficient. The system cannot be entirely threaded due to limitations within Unity, and certain bottlenecks which occur in the logic of the algorithm requiring it to go in a linear order preventing the threading of every execution; because of this, high-poly models may take seconds to shatter, which can freeze the render thread and interrupt gameplay. Figure 6 shows frame time of shattering between three sets of geometry, with material type glass (which has the most shatters) and concrete (which has a lower amount of shattering). The graph shows an exponential upwards tick in frame time, entering into the seconds, as the geometry becomes more complex.

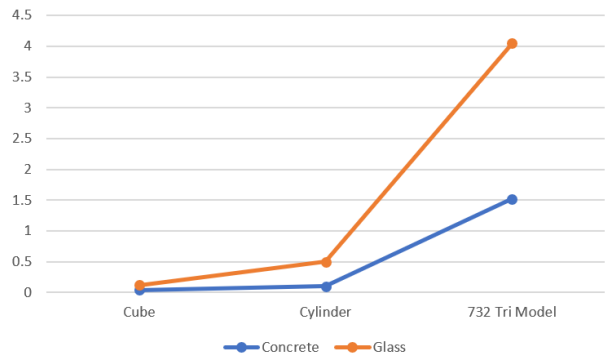

Figure 6: *A demonstration of frame times with different meshes.*

Although this implementation was sped up over development by implementing multithreading as best as possible and optimising some code, ultimately it remains limited to low poly uses. The system could be reworked to use processes similar to those described in the research section, for example, Voronoi patterns. Creating a pattern across a mesh using that as the cut line, rather than performing individual cuts one at a time, would likely be far more efficient.

5 Conclusion

While the system's performance can be poor for high poly geometry, its system of custom material types with physics properties is interesting and could be explored further. For low poly geometry the system works well, with convincing results, and non-noticeable impacts on performance. For that reason, the system at current could be carefully implemented into a project if used with small environment set pieces, rather than large scale high poly geometry.

Additionally, it would be good to add further material properties to the configuration tool, such as mass multipliers based on the object's in-world size and density; however the current offerings are suitable for the implementation at present. As suggested in the evaluation, Voronoi patterns could be applied to geometry to pre-compute shatters; if this was implemented it could also be configured in the material tool.

Bibliography

Chaos Destruction. URL: https://docs.unrealengine.com/en-US/Engine/Chaos/ChaosDestruction/index.html.

Grönberg, Anton (2017). "Real-time Mesh Destruction System for a Video Game". In:

Harrison, Linnea (2014). "Battlefield 4: Creating a More Dynamic Battlefield". GDC. URL: https://www.gdcvault.com/play/1020894/Battlefield-4-Creating-a-More.

Leadbetter, Richard (2019). "Crackdown 3 Wrecking Zone: what happened to the 'power of the cloud'?" In: URL: https://www.eurogamer.net/articles/digitalfoundry-2019-crackdown-3-wrecking-zone-what-happened-to-the-power-of-the-cloud.

Lien, Jyh-Ming and Nancy M. Amato (2006). "Approximate convex decomposition of polygons". In: *Computational Geometry* 35.1. Special Issue on the 20th ACM Symposium on Computational Geometry, pp. 100 –123. ISSN: 0925-7721. DOI: https://doi.org/10.1016/j.comgeo.2005.10.005. URL: http://www.sciencedirect.com/science/article/pii/S0925772105001008.

Müller, Matthias, Nuttapong Chentanez, and Tae-Yong Kim (July 2013). "Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions". In: *ACM Trans. Graph.* 32.4, 115:1–115:10. ISSN: 0730-0301. DOI: 10.1145/2461912.2461934.

URL: http://doi.acm.org/10.1145/2461912.2461934.

Parker, Eric G. and James F. O'Brien (2009). "Real-Time Deformation and Fracture in a Game Environment". In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '09. New Orleans, Louisiana: Association for Computing Machinery, 165–175. ISBN: 9781605586106. DOI: 10.1145/1599470.1599492. URL: https://doi.org/10.1145/1599470.1599492.