Advanced Technologies Task 2: Mesh Destruction and Deformation

Alex Feetham
17016942

University of the West of England

May 7, 2021

This report details the process I went through to implement a system capable of mesh destruction and deformation. The system needed to be able to slice a mesh and fill the hole created by the slicing.

1 Introduction

To best show an implementation of mesh destruction within a game-like environment, I decided to combine it with mesh deformation to create a realistic system which would cause debris to fracture from a wall when shot with by a projectile.

2 Related Work

Destruction has been around in video games for a long time. One of the earliest examples of destructible objects can be found in Space Invaders (Nishikado, 1978). The defence bunkers would break slightly each time they were shot until they were destroyed (Figure 1).

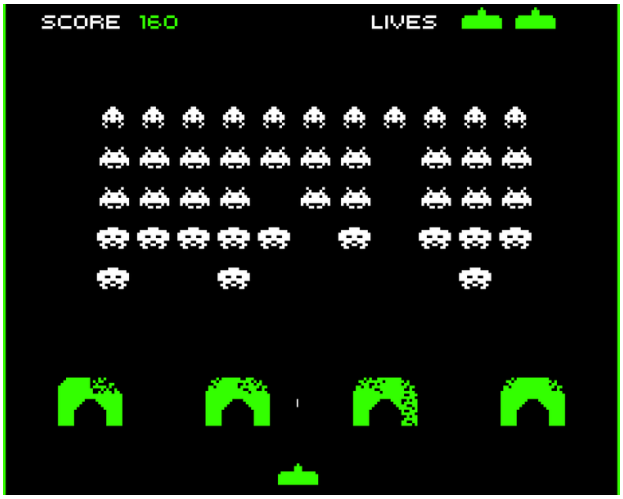

Figure 1: *Space Invaders*

Technology has come a long way since then and the hardware used in modern computers and games consoles are capable of creating much more realistic graphics and physics simulations. Triple-A titles of recent years have really been able to take advantage of this and have taken destructible environments to a whole new level. DICE's Battlefield franchise heavily utilises destructible environments. The player

can destroy almost anything, enemy hiding behind a wall? No problem, plant an explosive and blow it up; destroy enough of the structural supports to a building and the whole thing will collapse (Figure 2).

Figure 2: *A destroyed building in Battlefield 4*

Although technology has advanced a long way, mesh destruction is still very computationally taxing. One of the biggest issues with destroying 3-dimensional meshes is that they are hollow, so if you were to cut it in half, it would not be realistic as it has no inner volume. To simulate this, you would need to create a new face on the object at the point in which you cut it. This would need to be done for both halves of the object and the result would leave you with two different objects. To avoid this very heavy computation, (DICE, 2013) opt to use animation and pre-set events to destroy the environment. In their 2013 instalment of the franchise, Battlefield 4, many of the walls across different maps explode in exactly the same way. The walls have been pre-fractured and once a certain condition is met (i.e. explosive detonated in proximity to wall) then it will fracture in this predefined way, the debris will be the same and hole in the wall will be the same size and shape every time. This means the game does not need to compute anything special and can run this scripted event on this wall.

Figure 3: *A wall being blown up in Battlefield 4*

3 Method

Research into slicing meshes in Unity led to a mesh slicer implementation presented by (DitzelGames, 2019). The first phase of the destruction involves using planes to define where on an object to make the cuts. The planes are defined within the bounds of the object.

```
1 Plane plane = new Plane(UnityEngine.
    Random.onUnitSphere, new Vector3(
     UnityEngine.Random.Range(bounds.
    min.x, bounds.max.x), UnityEngine
    .Random.Range(bounds.min.y,
    bounds.max.y), UnityEngine.Random
    .Range(bounds.min.z, bounds.max.z
    )));
```

Once the plane is set, get all of the triangles in the mesh and check to see if they are on the left (negative) side of the plane. If any triangle has all of its vertices on the left side, then it is added to a new mesh. If a triangle has some vertices on either side of the plane, then this triangle needs to be cut. Using the Raycast function in Unity, the points at which the plane intersects the triangle can be found (Figure 4). There is a small problem in that both halves of the triangle need to join at the same point. To solve this, two rays are cast at exactly the same location, then Ray1 is used for the left side and Ray2 is used for the right side. These points are then connected to the vertices to the left side of the plane and are added to the new mesh. The vertices to the right (positive) side of the plane are also joined to the intersect points but these are kept to the original mesh. The original triangle has now been separated into three new triangles (Figure 5). MakeGameObject function is now called to set two halves of the split mesh as new game objects and to assign them with a mesh, mesh filter, collider and rigidbody. This process is repeated for the number of cuts specified. Figure 6 shows how this system affects a simple cube; it was set to destroy with 5 cutting planes.

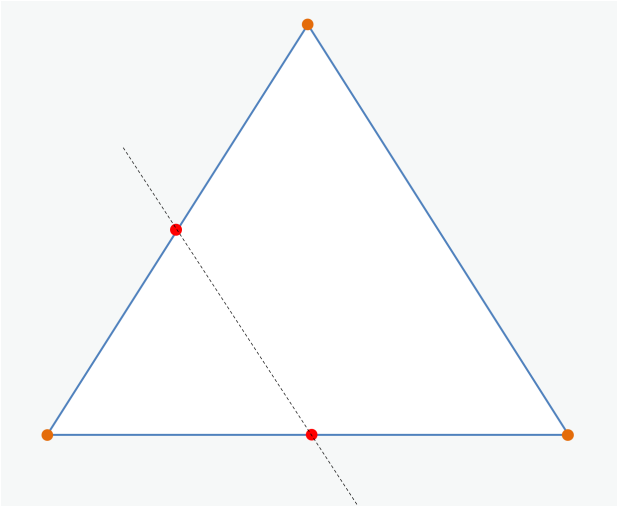

Figure 4: *Cutting plane intersecting a triangle*

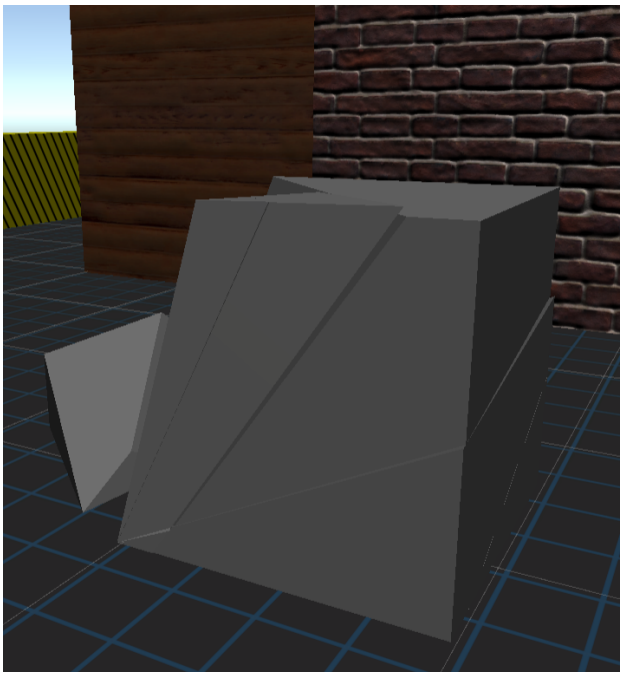

Figure 6: *Sliced Cube*

The second phase of the system is the deformation of the object that was shot. (Chapman, 2019) presents the following algorithm for deforming a mesh around a given point.

$$VertexPosition = Normal * Impulse * Scale * Malleability \quad (1)$$

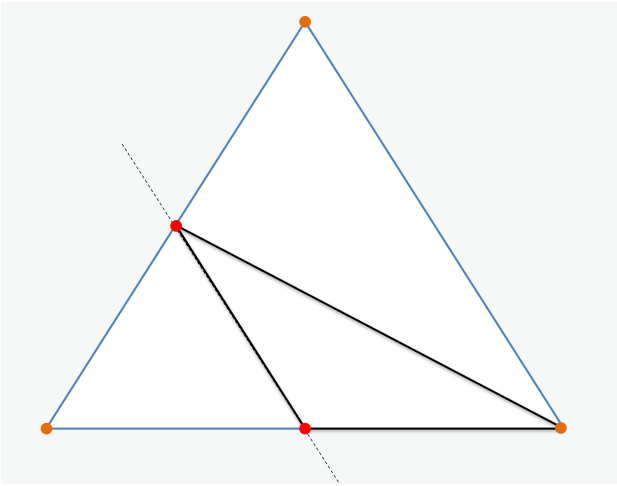

Figure 5: *New triangles created using intersect points as new vertices*

By combining the two systems a realistic simulation for shooting an object, indenting the object, and creating debris which fractures out from the impact point can be achieved. Figure 7 shows the result of the combined systems. The two walls have both been shot twice and small cubes were instantiated at the points of impact. The holes in the wood are wider and deeper as it was set with a greater radius and malleability than the bricks. As such, the initial debris has been scaled to fit the hole so as to observe the law of conservation of mass; “The same amount of matter exists before and after the change – none is created or destroyed (National Geographic Society, 2020). The wall objects also have a maximum tolerance of shots they can take, once this tolerance is reached, the wall will also break apart (Figure 8).

Figure 7: *2 walls that have been shot*

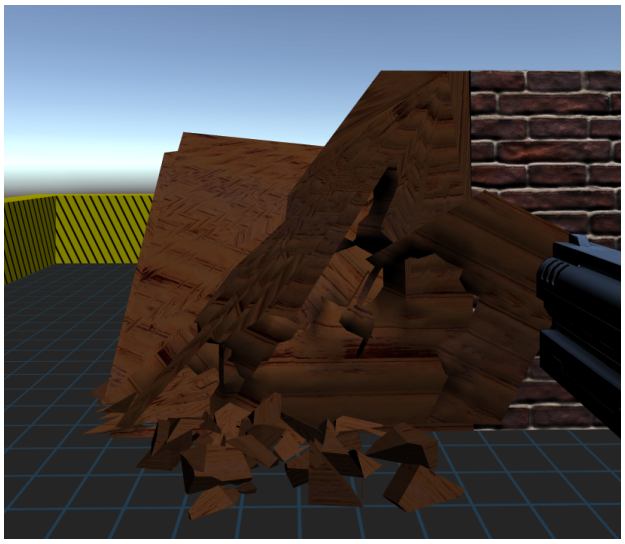

Figure 8: *Destroyed Wall*

This task has been extended by further expanding on the deformation aspect of the system. While the script created to deform a mesh on impact with a projectile worked well, it would not work for something such as creating tracks in snow. For this, a shader graph presented a more optimum solution. The shader graph can be used to create different materials so could be expanded to create materials for other ground types like sand or mud. The plane is deformed based on a map created by the trail of a particle system attached to the player. A camera is placed into the scene which observes only the particle system, the red channel of the camera is what is used to create the map. Figure 9 shows the tracks created in the snow where the red ball (player) has walked.

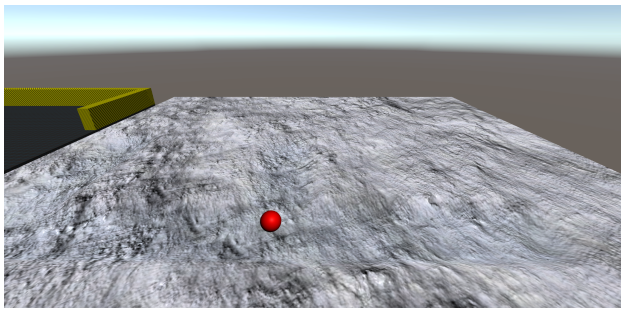

Figure 9: *Tracks in the snow*

4 Evaluation

I have made the system as optimised as possible so that it could be implemented within a game. Figure 10 shows the exposed properties for the script responsible for the system. It enables the user to toggle if they want the object to create debris in addition to show big the impact radius should be and the malleability of the object. Ranges have been imposed on the exposed parameters to optimise the system. If they were to exceed the bounds set, then it can be extremely detrimental to performance.

Figure 10: *Unity Inspector Tab*

The frames per second (FPS) of the program exceed 600FPS when the player is simply moving around the scene and firing the weapon. When the wall is hit and debris is spawned with the values set as shown in Figure 7, the program drops to between 120-250FPS for between 1-2 milliseconds (ms). When the wall is destroyed, the FPS drops to 15-25 for approximately 3ms. This is the best performance that was able to be extracted from the system on my personal desktop (Appendix B).

Prior to the optimisations, the FPS would drop to bellow 50FPS for over 5ms when the debris was instantiated and fragmented. This was because the debris fragments were inheriting the destruction script from the original game object that they are cloned from meaning that each fragment was then immediately recalculating its bounds and having all of the triangles stored in lists. By disabling the script for the debris fragments, this unnecessary computation

is no longer occurring and as such the performance of the system is increased. Additionally, the debris fragments have been set to automatically destroy themselves after 3 seconds. By doing so, it means there are less objects in the scene to save on processing power so the game can run smoother.

Another optimisation that was attempted was to enable threading. A thread represents part of a program which can be run independently (Vikram, 2004). The plan was to create several child threads and use them to execute the code for the fragmentation/destruction of the debris. Unfortunately, Unity does not allow for any properties regarding game objects to be modified outside of the main thread. This would mean that in order to enable threading on the program, the properties of the game object such as its position and list of triangles would need to be passed out of the main thread to the child thread where the calculations could be executed. The returned values from the child thread could then be used to update the game object back in the main thread. However, this means that the main thread would become hung as it awaits the child thread to return the values needed. Although the main thread would have been able to keep the game running at a steady FPS, it would not update the game objects within a reasonable/realistic time. Because of this issue, threading was not able to be implemented.

5 Conclusion

The system produced yields a good solution to the task. When a wall is struck by a projectile emitted from the player's weapon, the wall is deformed and debris fragments from the point of contact. The debris fragments are of random size while adhering to the law of conservation of mass. The system has been optimised to work in a user-friendly way so that it can be used in a plug-and-play fashion; all the system needs is for the DebrisBase prefab to be present in the scene and for the DefromableMesh script to be applied to an object. The debris automatically gets the appropriate material based on the object it is fragmenting off of and it appears realistic.

Bibliography

Chapman, Jordan (2019). *Mesh Deform*. GitHub. Accessed: 15/02/2021. URL: https://gist.github.com/Nordaj/33301ae473968c4fd0d096ffc427eb66.

DICE (2013). *Battlefield 4*. Electronic Arts. Accessed: 05/05/21. URL: https://www.ea.com/en-gb/games/battlefield/battlefield-4.

DitzelGames (2019). *Procedural Destroy in Unity*. YouTube. Accessed: 09/02/2021. URL: https://www.youtube.com/watch?v=VwGiwDLQ40A.

National Geographic Society, (2020). *The Conservation of Matter During Physical and Chemical Changes*. National Geographic. Accessed: 05/05/2021. URL: https://www.nationalgeographic.org/article/conservation-matter-during-physical-and-chemical-changes/6th-grade/.

Nishikado, Tomohiro (1978). *Space Invaders*. Atari. Accessed: 05/05/21.

Vikram, S (2004). *Software Development: The basics of threading in the .NET Framework*. Tech Republic. Accessed: 05/05/2021. URL: https://www.techrepublic.com/article/software-development-the-basics-of-threading-in-the-net-framework/.

Appendix A

Video Links:

1. Week 1
2. Week 2
3. Week 3
4.
5. Week 5

Appendix B

PC Specs:

- Ryzen 7 2700x CPU
- Nvidia RTX 2060 GPU
- 16GB Corsair RAM @3200mHz