The Procedural Destruction of Meshes

Connor Easterbrook
18015101

University of the West of England

March 22, 2023

Abstract

Using a modular design and multi-threading, this procedural mesh destruction project simulates mesh destruction and bodily dismemberment, presenting an accessible tool for external use. The calculations use computational geometry to decompose complex meshes into individual triangles, rendering the new textures correctly. The performance and effectiveness of the approach are evaluated through experiments on various test scenarios, with a concluding discussion on areas of improvement.

1 Introduction

1.1 Procedural mesh destruction

Procedural mesh destruction is an area of computer graphics that seeks to simulate the destruction of objects and characters in a dynamic and realistic manner. This project focuses on simulating body dismemberment and object shattering, using multi-threading to improve performance. When a video game is being developed, user interactivity is a big consideration that developers need to think about (Weber, Behr, and DeMartino, 2014). Pagan, 2019 suggests the interactivity and feedback of game objects enhance users' experience and game play immersion. One of the considered interactions would be whether an object can have its mesh manipulated, which often requires separate models for each effect. A procedural mesh destruction system would save a developer time, storage, and be relevant for every desired effect.

Procedural Mesh Destruction uses vertex-based calculations so that the end results comprise two full meshes with appropriate texturing, which can be seen in Figure 1 (Tobler, Maierhofer, and Wilkie, 2002). Morris and Anderson, 2010 outline a method that was referred to in this project that involves changing what resources the calculations use. Concerns arose in the beginning of the project about resource usage stemming from the calculations, which was solved through multi-threading via Unity's 'Job' system. This project was conducted in the Unity game engine as it provides various tools that assist in both user interactivity implementation and the creation and visualization of algorithms (Šmíd, 2017).

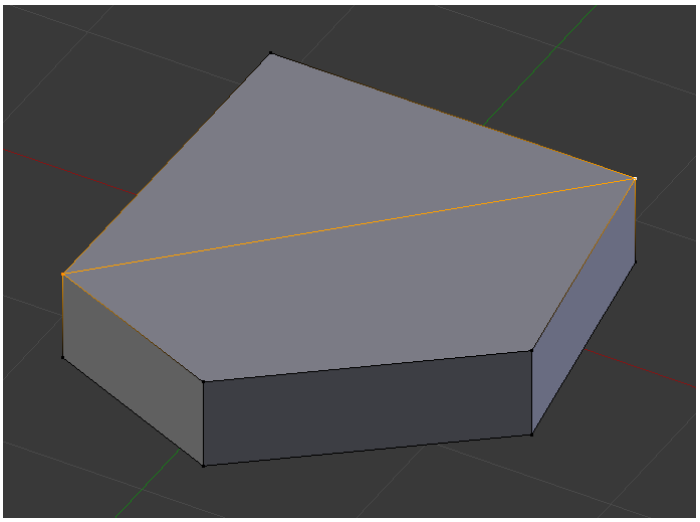

Figure 1: *Example of splitting a mesh. Previous slice on shape shown for reference.*

1.2 Related work

Tom Clancy's Rainbow Six Siege 2015 showcases procedural mesh destruction through every interactive wall being able to be destroyed, with varying results. An example of this can be seen in Figure 2. CatlikeCoding, 2015 published a tutorial on mesh deformation that was highly received by experts.

Figure 2: *Example of wall destruction in Tom Clancy's Rainbow Six Siege.*

1.3 Rationale

The potential applications for procedural manipulation of object meshes can range from entertainment to simulation (Grönberg, 2017). *Teardown* 2022 is a game that has procedural destruction be part of its core game play loop. Cutler et al., 2002 use fully procedural mesh deformation for simulations, while *Tom Clancy's Rainbow Six Siege* 2015 created a video game with destruction as a core component.

2 Methodology

As the calculations for procedural mesh destruction were programmed intending to be used in run-time, several additions were required to make the algorithm fit for usage (Lehtola, 2018). Among these additions were optimizations to allow for calculations to cause minimal performance loss. An example of this would be the multi-threading that is conducted through the Unity Game Engine's 'Jobs' system, which is detailed by Borromeo, 2020, as an appropriate method of optimization.

This mesh destruction algorithm can be dissected into three core sections: event, calculation, and result. 'Event' specifies the initial event that calls the algorithm and begins the calculations specified in the 'Calculation' section. 'Result' details the subsequent meshes and steps taken after calculations have been completed.

2.1 Event

To trigger the mesh destruction calculations, a collision event must first occur where the mesh receives a collision input that calls the destruction script. There are a few procedural implementations for this, but the most simple, and the one used within this project, is using a slice plane. This method was chosen as it allowed for the exploration of multiple methods of mesh destruction because of the minimal programming required (Van Gestel, 2011).

A slice plane is a plane that intersects a mesh and divides it into two parts. The mathematical formula for finding the slice plane of a mesh will depend on the specific representation of the mesh and the desired properties of the slice plane. The formulas that appear in this project are:

1. If the mesh is represented as a collection of vertices and faces, and the slice plane is defined by a normal vector and a point on the plane (e.g. a bullet collision), the formula for finding this slice plane would be:

$$Ax + By + Cz + D = 0 \tag{1}$$

2. If the mesh is represented as a collection of vertices and faces, and the slice plane is defined by the cross product of two vectors (e.g. a sword slice), the formula for finding this slice plane would be:

$$C = AxB = |A| * |B| * sin(z) * n \tag{2}$$

Where |A| and |B| are the magnitudes of A and B, z is the angle between A and B, and n is a point perpendicular to both A and B.

2.2 Calculation

Immediately after an event trigger is called to destroy the mesh, the calculations begin and happen in a set order - separation, filling, instantiation. Optimizations of these scripts have strongly followed Dickinson, 2017's work.

During the separation part of the calculation, the mesh's information is stored with unity's Gameobject class. The mesh then runs through a set of equations that establish the locations of vertices, normals, and UVs, which will be referred to as triangles (Stegmayr, 2008). These triangles are examined through an iterating loop and checked to see if the triangle is intersecting with the slice plane. This is done using the multi-threading "Jobs" system in Unity, allowing the calculation to be conducted on multiple threads for a performance increase (Morris and Anderson, 2010). Borufka, 2020 provided inspiration for where and how to implement this. If the triangle does not intersect with the slice plane, then it is stored in a class alongside the mesh data, otherwise it examines calculations to correctly slice it. Triangle slicing is calculated by finding on which side of the slice plane the vertices are and adding new vertices on the slice plane point to fill the gap caused. These triangles are then tested to see if they are correctly placed, and if so, then added to the mesh data.

The filling of new triangles is relatively simple because it runs through the vertices, evaluates whether the triangle faces outwards and is valid, then fills in the gap. An extra check is run to check the direction of the triangle face to ensure that it is facing the correct way. The calculation of this can be seen in the below code.

```
1  private int NextSlot()
2  {
3    if (Vector3.Dot(Vector3.Cross(_vertices[1] -
       _vertices[0], _vertices[2] - _vertices[0]),
       _normals[0]) < 0)
4    {
5      Vector3 temp = fillTriangle.vertices[2];
6      fillTriangle.vertices[2] = fillTriangle.
       vertices[0];
7      fillTriangle.vertices[0] = temp;
8
9      temp = fillTriangle.normals[2];
10     fillTriangle.normals[2] = fillTriangle.normals
       [0];
11     fillTriangle.normals[0] = temp;
12   }
13
14   mesh.AddTriangle(fillTriangle);
15 }
```

Regarding finding which triangles, made from connected vertices and faces, are intersecting with the slice plane, the following calculations take place:

Let P_1 and P_2 be the two points that define the cutting plane for the mesh slice. Let T be the set of triangles in the original mesh that intersect the cutting plane. For each triangle $t_i \in T$, calculate the intersection point I_i of the triangle's plane and the cutting plane:

$$I_i = \frac{(P_2 - P_1) \cdot (P_{1i} - P_1)}{(P_2 - P_1)^2}(P_2 - P_1) + P_1 \tag{3}$$

where P_{1i} is one of the vertices of t_i.

Then, create two new triangles: one with vertices P_1, I_i, and the third vertex of t_i, and the other with vertices P_2, I_i, and the third vertex of t_i:

$$t_{1i} = (P_1, I_i, P_{3i}) \tag{4}$$

$$t_{2i} = (P_2, I_i, P_{3i}) \tag{5}$$

Finally, assign a rendered texture to the newly created faces.

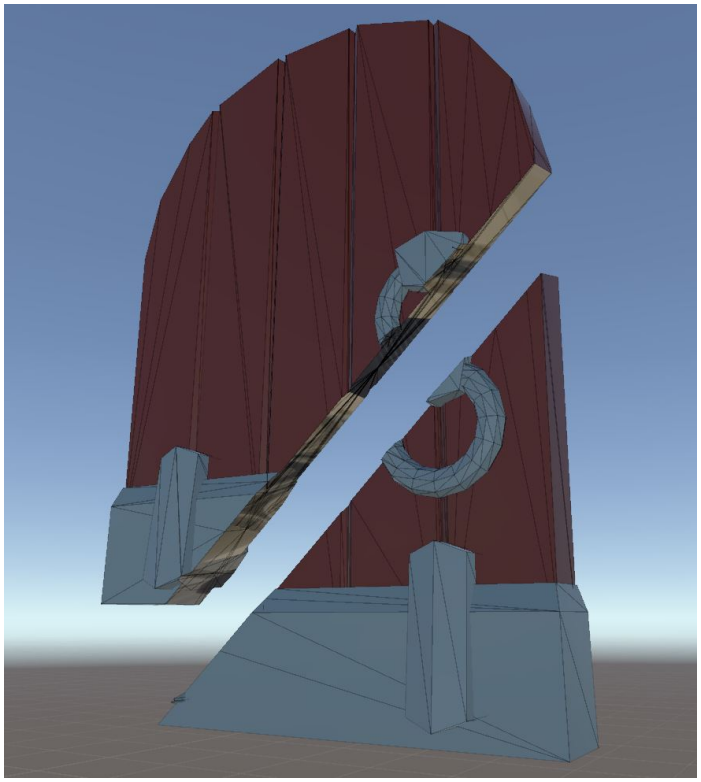

Figure 3: *Showcase of a complex sliced mesh. Unique fill and wire frame view allows to see the newly formed triangles.*

Upon completing all calculations to for the two new meshes made from the original mesh, the instantiation begins. This instantiation is done by creating a new object and assigning all the stored mesh information to each one, respectively. There is more information in the next subsection but an example can be seen in Figure 3.

2.3 Result

The result of calculating the mesh destruction is formed by combining all stored data along with adding components that may be required in order for the new meshes to simulate the original Gameobject. This is done by reading the original Gameobject data and running through its components, adding what was on it to the new objects. Texturing is then calculated by copying over the Mesh Renderer component values and assigning the new triangles to mimic a previously assigned texture. These combine to create a seamless mesh destruction simulation that works during runtime.

3 Evaluation

Overall, this project is a standalone tool that showcases efficient procedural destruction. Using procedural techniques and multi-threading to simulate body dismemberment and object shattering is noteworthy, as it allows for more realistic and dynamic destruction effects to be generated. However, the usability and performance of the project may require careful consideration and optimization, depending on the specific application and intended user audience.

3.1 Usability

The usability of this project, within other projects, depends on the ease of implementation and its ability to be integrated into existing environments. While the project's modular design and descriptive process allow for an easier experience for implementation, the accessibility may be limited by the user's technical knowledge. Therefore, the usability of this project as a tool may be better suited for experienced users.

3.2 Performance

The performance of a procedural mesh destruction project can have a significant impact on overall performance. The multi-threaded and modular design of this project is promising in terms of performance, as it distributes processing across multiple cores and improve simulation efficiency, as seen in Figures 4 and 5. These figures also represent the improvement of performance in game-like situations, which is important. However, the complexity of simulating body dismemberment and object shattering can still be computationally intensive and require significant resources. Further optimization and testing will likely be necessary to increase performance to the point of this project's performance hit to a software system be unnoticeable to the end user.

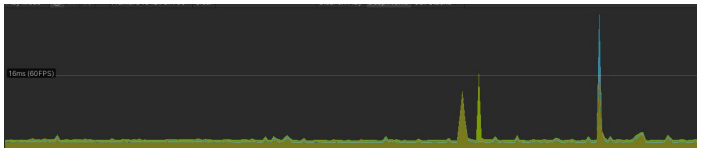

Figure 4: *Showcase of mesh slicing before multi-threading was implemented. The spike lowered the frame rate of the application to 15 on calculation, from above 500. The base frame rate was higher due to base lighting.*

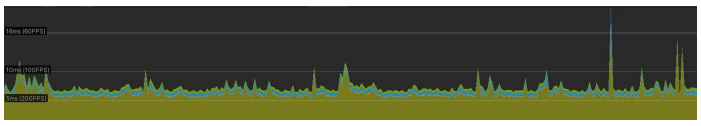

Figure 5: *Showcase of mesh slicing after multi-threading was implemented. The spike lowered the frame rate of the application to 45 on calculation, from above 250. The base frame rate was lower due to real-time lighting.*

4 Conclusion

The system produced yields a good solution to the task specifications. A mesh is procedurally destroyed upon request and the user is able to explore this in various ways. A form of dismemberment has been implemented within the project to allow for limbs to also be procedurally destroyed, allowing for more variety in a game play experience. The system has been multi-threaded so that it functions without being overly resource-intensive and unobtrusive to the user's experience. The final result of this project forms a simple library that allows for easy implementation of mesh destruction in any Unity Game Engine project. Further development could be done through complicating the event portion of the mesh destruction calculations, allowing for more realistic results through a more realistic destruction path. An example of this would be utilizing a Lindenmayer system to generate unique destruction paths.

In conclusion, it is believed that this project achieves its goal of being a procedural mesh destruction system in an appropriate manner, but there is room for improvement.

Bibliography

Borromeo, Nicolas Alejandro (2020). *Hands-On Unity 2020 Game Development: Build, customize, and optimize professional games using Unity 2020 and C*. Packt Publishing Ltd.

Borufka, Roman (2020). "Performance testing suite for Unity DOTS". In.

CatlikeCoding (2015). *Mesh deformation, a unity C tutorial*. Catlike Coding. URL: https://catlikecoding.com/unity/tutorials/mesh-deformation/.

Cutler, Barbara et al. (2002). "A procedural approach to authoring solid models". In: *ACM Transactions on Graphics (TOG)* 21.3, pp. 302–311.

Dickinson, Chris (2017). *Unity 2017 Game Optimization: Optimize All Aspects of Unity Performance*. Packt Publishing Ltd.

Grönberg, Anton (2017). *Real-time mesh destruction system for a video game*.

Lehtola, Aleksi (2018). "Optimizing Unity Projects". In.

Morris, Derek John and Eike Falk Anderson (2010). "GPU Destruction: Real-Time Procedural Demolition of Virtual Environments." In: *Eurographics (Posters)*.

Pagan, Elizabeth (2019). *Designing interactivity into game play*. URL: https://usv.edu/blog/designing-interactivity-into-game-play/.

Šmíd, Antonín (2017). "Comparison of unity and unreal engine". In: *Czech Technical University in Prague*, pp. 41–61.

Stegmayr, Christofer (2008). "Procedural deformation and destruction in real-time". In.

Teardown (2022). Steam. Tuxedo Labs.

Tobler, Robert F, Stefan Maierhofer, and Alexander Wilkie (2002). "A multiresolution mesh generation approach for procedural definition of complex geometry". In: *Proceedings SMI. Shape Modeling International 2002*. IEEE, pp. 35–271.

Tom Clancy's Rainbow Six Siege (2015). Steam. Ubisoft Montreal.

Van Gestel, Joris (2011). "Procedural destruction of objects for computer games". In: *Delft University of Technology*.

Weber, René, Katharina-Maria Behr, and Cynthia DeMartino (2014). "Measuring interactivity in video games". In: *Communication Methods and Measures* 8.2, pp. 79–115.

Appendix A

Week 1 - https://youtu.be/Fnkc7f2bV44
Week 2 - https://youtu.be/0zBLTsGmVO8
Week 3 - https://youtu.be/F_cUZE7dwWw
Week 4 - https://youtu.be/pRW35199xXc
Week 5 - https://youtu.be/0tgC69qQnVE