Real-time Fracture in Unity

DENIS ALEXANDRE RACICOT MORALES, McGill University

This paper examines a method to compute object fractures in real time. The technique uses stress and separation tensors computed over finite elements to determine whether a crack should occur and how it should break the mesh. The goal is to present the math in a digestible way such that anyone reading can understand the principles and create their own version afterward. We then present and test a working implementation in Unity. Finally, we evaluate its performance and hint at how it could be further improved.

Code: https://github.com/Seibaah/Fracture
Video: https://youtu.be/HzDzOsPxKlg

Additional Key Words and Phrases: fracture, finite elements, real-time, 3D, Unity

ACM Reference Format:
Denis Alexandre Racicot Morales. 2023. Real-time Fracture in Unity. 1, 1 (April 2023), 3 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

As hardware performance and physically based animation techniques improve, the desire to create more complex and realistic experiences increases. The evolution of video games over the last 20 years is a testament to this. We went from being limited to drawing only a handful of sprites on screen to supporting real-time ray tracing. One of the examples of this unprecedented leap is the ability to simulate real-time physically-induced object destruction.

This paper describes the math behind quick and realistic-looking fractures based on existing literature and presents a working implementation in Unity. Finally, we pit this implementation against different scenarios that will test the limit of its capabilities and decide if it can produce fractures in real time.

2 RELATED WORK

O'Brien et al. [1999] were the earliest reference we studied for this paper. They improve contemporary techniques and simulated objects' brittle fractures by analyzing finite element stress tensors. They formulate the stress tensor of an isotropic material using Green Strain. Using the forces tetrahedral finite elements exert on their nodes and the stress tensor, they compute a separation tensor. If the eigenvalues of the separation tensor exceed the material threshold parameter, fracture occurs in the direction orthogonal to the corresponding eigenvectors. Because the focus is on producing an accurate simulation, they also discuss re-meshing when the fracture plane cuts through tetrahedra. While this technique produces accurate results, it is expensive and does not run in real-time. Thus, this is unsuitable for a video game.

Parker et al. [2009] address this and propose a simplified version of the same algorithm that can run in real time and provide satisfying results. They replace the Green strain tensor with the easier-to-compute Cauchy strain tensor. However, because Cauchy strain is not invariant to rotation, they perform a polar decomposition to create a corotational strain. They then compute force decomposition and the separation tensor like O'Brien et al. [1999] did. No re-meshing is done. Instead, Parker et al. [2009] propose using a finer representation of the mesh called splinters to hide the fact that the underlying fracture can only occur along tetrahedral edges. Overall, the result is a balanced approach that combines realistic-looking fractures and efficiency, and video games such as Star Wars: The Force Unleashed have adopted the method.

Author's address: Denis Alexandre Racicot Morales, McGill University, denis.racicotmorales@mail.mcgill.ca.

2023. XXXX-XXXX/2023/4-ART $15.00
https://doi.org/0000001.0000001

3 METHODS

3.1 Finite Element Discretization

We use a finite element method to model the stress materials are under during the simulation. Our method employs non-degenerate tetrahedral elements where each node (vertex) is defined by world positions $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3$, and $\boldsymbol{u}_4$. Let $\boldsymbol{Du}$ be 3×3 matrix with columns $\boldsymbol{u}_2 - \boldsymbol{u}_1, \boldsymbol{u}_3 - \boldsymbol{u}_1$, and $\boldsymbol{u}_4 - \boldsymbol{u}_1$. The element basis is then $\boldsymbol{\beta} = \boldsymbol{D}_u^{-1}$. Additionally, define a matrix $\boldsymbol{D}_x$ matrix similar to $\boldsymbol{D}_u$, such that $\boldsymbol{x}_i$ is the current node's position. The deformation gradient is given by

$$\boldsymbol{F} = \frac{\partial x}{\partial u} = \boldsymbol{D}_x \boldsymbol{\beta}$$

Cauchy's infinitesimal strain tensor $\boldsymbol{\varepsilon} = 1/2\left(\boldsymbol{F} + \boldsymbol{F}^\top\right) - \boldsymbol{I}$ is cheap to compute from $\boldsymbol{F}$. However, because it is not invariant with respect to rotation, we must factor out the rotation per element. To do so, we perform a polar decomposition of $\boldsymbol{F}$ into $\boldsymbol{F} = \boldsymbol{QA}$ where $\boldsymbol{Q}$ is orthonormal, and $\boldsymbol{A}$ is symmetric. In practice, we obtained the polar decomposition through the Singular Decomposition of $\boldsymbol{F}$ into $\boldsymbol{F} = (\boldsymbol{V}\Sigma\boldsymbol{V}^T)(\boldsymbol{W}\boldsymbol{V}^T) = \boldsymbol{UP} = \boldsymbol{QA}$

With $\boldsymbol{Q}$ computed, we can replace $\boldsymbol{F}$ with $\tilde{\boldsymbol{F}} = \boldsymbol{Q}^\top \boldsymbol{F}$. The rotational invariant (corotational) strain is given by $\tilde{\boldsymbol{\varepsilon}} = 1/2\left(\tilde{\boldsymbol{F}} + \tilde{\boldsymbol{F}}^\top\right) - \boldsymbol{I}$. Because we assume the materials are isotropic, the stress tensor of the element is given by $\boldsymbol{\sigma} = \lambda \operatorname{Tr}(\tilde{\varepsilon})\boldsymbol{I} + 2\mu\tilde{\varepsilon}$, where λ and μ are the material's Lamé parameters.

The elastic force exerted by the tetrahedral element on its nodes is $\boldsymbol{f}_i = \boldsymbol{Q}\boldsymbol{\sigma}\boldsymbol{n}_i$, where $\boldsymbol{n}_i$ is the area-weighted normal of the face opposite to node i.

3.2 Force Decomposition

The forces acting on a node can be decomposed by separating the stress tensor into tensile and compressive components. To do so, first, O'Brien et al. [1999] define the following:

$$\boldsymbol{\beta} = \begin{bmatrix} \boldsymbol{m}_{[1]} & \boldsymbol{m}_{[2]} & \boldsymbol{m}_{[3]} & \boldsymbol{m}_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1}$$

$$\mathbf{m}(\boldsymbol{a}) = \begin{cases} \boldsymbol{a}\boldsymbol{a}^\top/|\boldsymbol{a}| & : \quad \boldsymbol{a} \neq \boldsymbol{0} \\ \boldsymbol{0} & : \quad \boldsymbol{a} = \boldsymbol{0} \end{cases}$$

$\boldsymbol{\beta}$ is the inverse of the 4×4 matrix with the material coordinates $\boldsymbol{m}_{[i]}$ of each node i as columns. $\mathbf{m}(\boldsymbol{a})$ is an operator that takes a

vector $\boldsymbol{a}$ in $\Re^3$ and constructs a 3×3 symmetric matrix that has $|\boldsymbol{a}|$ as an eigenvalue with $\boldsymbol{a}$ as the corresponding eigenvector, and with the other two eigenvalues equal to zero.

Let $v^i(\boldsymbol{\sigma})$ is the ith eigenvalue of $\boldsymbol{\sigma}$ and $\hat{\mathbf{n}}^i(\boldsymbol{\sigma})$ its corresponding unit length eigenvector. Now, the tensile and component $\boldsymbol{\sigma}^+$ can be calculated by:

$$\boldsymbol{\sigma}^+ = \sum_{i=1}^{3} \max\left(0, v^i(\boldsymbol{\sigma})\right) \mathbf{m}\left(\hat{\mathbf{n}}^i(\boldsymbol{\sigma})\right)$$

Let $\boldsymbol{p}_{[i]}$ be the tetrahedral ith node position in world coordinates. Then, the tensile forces can be computed by:

$$f^+_{[i]} = -\frac{\text{vol}}{2} \sum_{j=1}^{4} \boldsymbol{p}_{[j]} \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl}\beta_{ik}\sigma^+_{kl}.$$

The compressive component and forces can be computed similarly, but because $\boldsymbol{\sigma} = \boldsymbol{\sigma}^+ + \boldsymbol{\sigma}^-$, and $\boldsymbol{f}_{[i]} = \boldsymbol{f}^+_{[i]} + \boldsymbol{f}^-_{[i]}$ both can be computed more efficiently by subtraction.

Now each node has a set of tensile and compressive forces that are exerted by the elements attached to it. For a given node, these sets are $\{\boldsymbol{f}^+\}$and $\{\boldsymbol{f}^-\}$respectively. We also keep track of the sum over each set as $\boldsymbol{f}^+$ and $\boldsymbol{f}^-$.

3.3 The Separation Tensor

The separation tensor describes the forces acting at each node and is used to determine if a fracture should occur at that point. It is computed as follows:

$$\boldsymbol{\varsigma} = \frac{1}{2}\left(-\mathbf{m}\left(\boldsymbol{f}^+\right) + \sum_{\boldsymbol{f}\in\{\boldsymbol{f}^+\}} \mathbf{m}(\boldsymbol{f}) + \mathbf{m}\left(\boldsymbol{f}^-\right) - \sum_{\boldsymbol{f}\in\{\boldsymbol{f}^-\}} \mathbf{m}(\boldsymbol{f})\right)$$

Let v^+be the largest positive eigenvalue of $\boldsymbol{\varsigma}$. If $v^+$is greater than the material toughness, τ, the material will break at the node, and the fracture direction will be perpendicular to the corresponding eigenvector of v^+.

4 IMPLEMENTATION

We chose Unity for our implementation as it is a powerful and versatile game engine suitable for real-time rendering. Thanks to its extensive functionality, we were able to focus on developing the fracture implementation itself without worrying about building collision detection or time stepping first. Our simulations use colliders and rigid bodies to detect when a projectile hits a breakable mesh and start the fracture algorithm. Unity lacks advanced mathematical computation capabilities. So we paired it with MathNET Numerics to handle the computations due to its comprehensive linear algebra functions and ease of use.

We need a tetrahedral representation of our meshes (see Fig. 1). It is acceptable to precompute this. We used PyVista and TetGen to save the tetrahedral mesh decompositions on the disk, and when a scene starts, load the files.

When started, the fracture algorithm runs sequentially on all the mesh tetrahedra and computes the set of tensile and compressive forces acting on each node. Then, each node computes its separation tensor, and if the maximum eigenvalue exceeds the material threshold, we fracture the mesh.

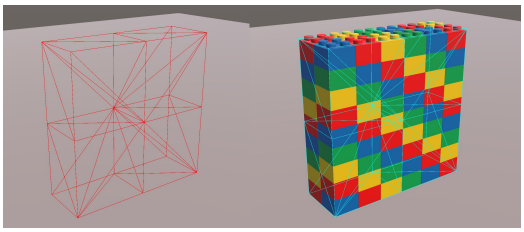

Fig. 1. Tetrahedral mesh wireframe only (left) and tetrahedral mesh with splinters (right)

Fracturing involves splitting the mesh along a plane normal to the corresponding eigenvector, duplicating shared nodes, and updating vertices and tetrahedra for each side. Additionally, mesh boundaries must be re-certified to avoid unrealistic connections, such as subsections connected only by one or two vertices. We must split the mesh again if the boundary cannot be certified.

Mesh splitting is expensive, even without re-meshing. Thus, we set heuristics to curb the algorithm runtime. We set a maximum number of fracture events allowed per frame such that we stop processing fractures even if more should occur in the current frame. Boundary re-certification also counts towards the maximum fracture number since it can split the mesh. Finally, we also limit the time spent in mesh re-certification.

5 RESULTS

We tested our implementation against three different test suites. In each test, we create a destructible mesh when the scene starts, and after some time, a script fires a burst of 3 projectiles toward the mesh. We recorded the simulation using the Recorder package. The results can be seen in the video attached to the paper. It is worth noting that the FPS performance while recording is lower by about 1.5x-4x, depending on the test case.

The first test suite measures performance and correctness vs. mesh complexity. It consists of three tests with a single mesh instance of increasing size and tetrahedral complexity. The algorithm proved capable of handling with ease small and medium meshes (<100 tetrahedra) (see Fig. 2). On the other hand, the large mesh test (438 tetrahedra) suffered from stuttering when the projectiles impacted it for the first time. The fact that the stutters happen only in the first collision suggests that dividing a mesh with a high number of tetrahedra is too expensive.

We want our algorithm to handle multiple breakable mesh instances in the scene. The second testing suite evaluates this. We fractured sets of 4, 16, and 32 small meshes concurrently. Performance throughout was excellent. Interestingly, although the 32-instance test contains more tetrahedra than the large mesh test from the previous suite, the former did not suffer from stuttering. This observation supports our theory that our implementation struggles with the high complexity of meshes rather than the scene's total tetrahedral count.

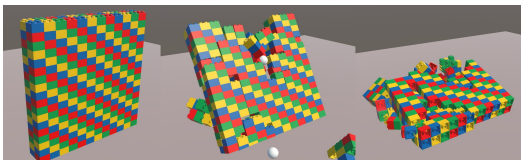

Fig. 2. From left to right, medium mesh destruction test before impact, just after impact, and in its final resting state

Finally, we want our implementation to be able to model different materials and their resistance (or lack thereof) to fracture. We created a test with three instanced small meshes with varying Young modulus, Poisson ratio, toughness threshold, and mass parameters. As expected, the meshes' damage is inversely proportional to the strength of their parameters.

Overall, our implementation can handle real-time fracture for different resolution meshes and produce satisfying-looking results. It handles small and medium complexity meshes exceptionally well. On the other hand, it becomes unstable against large meshes with high tetrahedral complexity.

Future work should focus on improving stability and implementing concurrency. Reworking the implementation to compute the acting forces and separation tensors using multithreading would be an exciting challenge. Alternatively, reducing memory use while maintaining real-time performance is another equally attractive proposition.

6 CONCLUSIONS

Combining previous literature work, we have presented a method suitable for real-time fracture computation. We explained the math that helps us determine, according to the physical properties of a material, if and how an object should break under stress. We then gave a high-level overview of the technique implemented in Unity. Finally, we verified that its behavior and performance make it suitable for generating realistic-looking fractures in real time. The reader should now have the tools to improve the presented implementation or build their own should they want to.

REFERENCES

[1] J. F. O'Brien and J. K. Hodgins, "Graphical Modeling and Animation of Brittle Fracture," in Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, 1998, pp. 137-146.

[2] E. Parker and J. F. O'Brien, "Real-time deformation and fracture in a game environment," in Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, 2009, pp. 119-126.