Real-Time tearing and fracturing

Myriam Beauvais*
ID 260760034
COMP-559, Fundamentals of Computer Animation

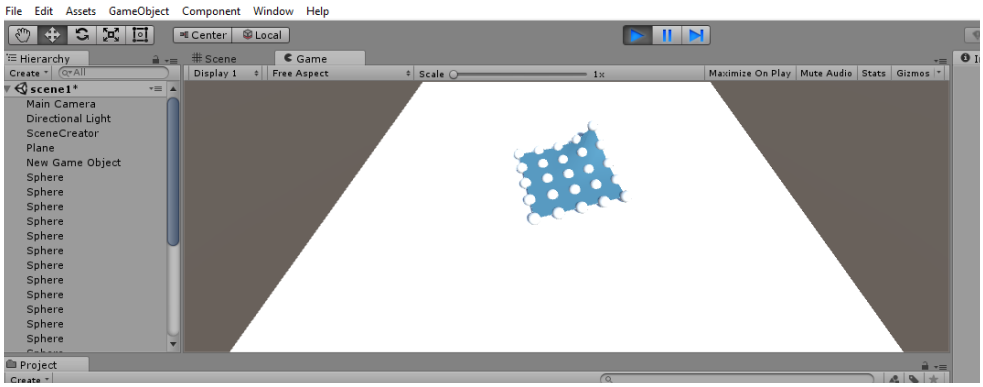

Figure 1: *In-Game view of the Project's scene and interaction*

1 Abstract

This project is an attempt at replicating [O'Brien and Hodgins 1999]'s cracking and fracturing work in Unity 3D, leveraging its *PhysX* physics engine. The work relies on calculating stress tensors for the different triangles of a mesh and analyse the corresponding forces to assess if the object will fracture and where and how it will do so. Based on the latter information, remeshing of the object is executed. For simplification, our implementation only handles 2D meshes, allowing to do the tensor calculations on triangles instead of on tetrahedrons.

2 Introduction

A lot of different objects, when under some persistent and or increasing forces, will end up getting distorted and breaking apart. Because virtual objects are often put in such situations in video games, movies or simulations, addressing this topic is very pertinent and interesting in order to recreate realistic sequences.

The project presented is the implementation of an interactive application allowing for deformation, tearing and fracturing of meshes in real-time, according to the method described in [O'Brien and Hodgins 1999].

The initial proposal was to implement the work of [Parker and O'Brien 2009]. While the overall work adds to the fracturing algorithm, the section specifically dedicated to fracturing is very similar to [O'Brien and Hodgins 1999]'s method except that they don't replicate tetrahedrons while remeshing. To focus more on the fracturing algorithm, we worked on [O'Brien and Hodgins 1999]'s method exclusively.

*e-mail:myriam.beauvais@mail.mcgill.ca

3 Related Work

Fracturing of objects is a well studied topic for which many approaches were proposed. While many models are based of physically based methods, some also proposed less physically perfect but visually appealing techniques such as procedural algorithms.

This work is mainly based of the algorithm described by [O'Brien and Hodgins 1999] which models a fracture using a separation tensor, calculated over the volume of a mesh. They calculate the separation tensors of a point based on the force tensors of connecting tetrahedral elements.

[Parker and O'Brien 2009] Propose improvements upon [O'Brien and Hodgins 1999], optimizing and adapting its work, using many ideas proposed in the literature, in order to provide a system that simulates fractures and deformations in a real-time context.

4 Technical details

As this project is mainly the implementation of [O'Brien and Hodgins 1999] for a 2D mesh in Unity, most of the technical details revolves around adapting the algorithm to the Unity context and how to leverage Unity's physics Engine to handle collisions and most physics aspects unrelated directly to fracturing. To understand better the work that has been made, we'll describe the modifications made to [O'Brien and Hodgins 1999]'s algorithm and the technical aspects needed to implement it in Unity.

4.1 Fracturing

Most of the calculations made in this project are in order to compute the separation tensor. Using the latter gives information about if a fracture should happen, at which point and how it will spread.

Based on a simplification of [O'Brien and Hodgins 1999]'s fracturing algorithm, we first divide our 2D mesh into triangle elements, each defined by three nodes. The latter have a position in material coordinates, $\boldsymbol{m}$ - which in Unity corresponds to the mesh's *GameObject*'s local space - and a position and a velocity in world space, $\boldsymbol{p}$ and $\boldsymbol{v}$.

We define three matrices $\mathbf{P}$, $\boldsymbol{V}$ and $\boldsymbol{\beta}$ as

$$\boldsymbol{P} = [\boldsymbol{p}_{[1]} \quad \boldsymbol{p}_{[2]} \quad \boldsymbol{p}_{[3]}] \tag{1}$$

$$\boldsymbol{V} = [\boldsymbol{v}_{[1]} \quad \boldsymbol{v}_{[2]} \quad \boldsymbol{v}_{[3]}] \tag{2}$$

$$\boldsymbol{\beta} = [\boldsymbol{m}_{[1]} \quad \boldsymbol{m}_{[2]} \quad \boldsymbol{m}_{[3]}]^{-1}. \tag{3}$$

These are used to build two 3×3 tensors; the Green's strain tensor, $\boldsymbol{\epsilon}$ and the strain rate tensor $\boldsymbol{\nu}$ defined by

$$\epsilon_{ij} = \left(\frac{\partial \boldsymbol{x}}{\partial u_i} \cdot \frac{\partial \boldsymbol{x}}{\partial u_j} \right) - \delta_{ij}, \tag{4}$$

$$\nu_{ij} = \left(\frac{\partial \boldsymbol{x}}{\partial u_i} \cdot \frac{\partial \dot{\boldsymbol{x}}}{\partial u_j} \right) + \left(\frac{\partial \dot{\boldsymbol{x}}}{\partial u_i} \cdot \frac{\partial \boldsymbol{x}}{\partial u_j} \right) \tag{5}$$

with $\boldsymbol{u} = [u, v, w]^T$, a location in the material coordinate, $\boldsymbol{x(u)} = [x, y, z]^T$ a function mapping the material coordinates to world coordinates and δ_{ij} being the Kronecker delta :

$$\delta_{ij} = \begin{cases} 1 : i = j \\ 0 : i \neq j. \end{cases} \tag{6}$$

The tensors can be easily computed using

$$\frac{\partial \boldsymbol{x}}{\partial u_i} = \boldsymbol{P}\boldsymbol{\beta}\boldsymbol{\delta}_i \tag{7}$$

$$\frac{\partial \dot{\boldsymbol{x}}}{\partial u_i} = \boldsymbol{V}\boldsymbol{\beta}\boldsymbol{\delta}_i \tag{8}$$

with

$$\boldsymbol{\delta}_i = [\delta_{i1} \quad \delta_{i2} \quad \delta_{i3}]^T. \tag{9}$$

We then calculate the elastic stress $\boldsymbol{\sigma}^{(\epsilon)}$, function of the strain, the material's rigidity μ and dilation λ and the viscous stress $\boldsymbol{\sigma}^{(\nu)}$, function of the strain rate and two parameters ϕ and ψ :

$$\boldsymbol{\sigma}_{ij}^{(\epsilon)} = \sum_{k=1}^{3} \lambda \epsilon_{kk} \delta_{ij} + 2\mu\epsilon_{ij} \tag{10}$$

$$\boldsymbol{\sigma}_{ij}^{(\nu)} = \sum_{k=1}^{3} \phi \nu_{kk} \delta_{ij} + 2\psi\nu_{ij}. \tag{11}$$

The total resulting stress $\boldsymbol{\sigma}$ is the sum of the elastic and the viscous stress, $\boldsymbol{\sigma} = \boldsymbol{\sigma^{(\epsilon)}} + \boldsymbol{\sigma^{(\nu)}}$. The stress tensor is then decomposed into a tensile component, $\boldsymbol{\sigma}^+$, and a compressive component, $\boldsymbol{\sigma}^-$:

$$\boldsymbol{\sigma}^+ = \sum_{i=1}^{3} \max(0, \mathrm{v}^i(\boldsymbol{\sigma}))\, \mathbf{m}(\hat{\mathbf{n}}^i(\boldsymbol{\sigma})) \tag{12}$$

$$\boldsymbol{\sigma}^- = \sum_{i=1}^{3} \min(0, \mathrm{v}^i(\boldsymbol{\sigma}))\, \mathbf{m}(\hat{\mathbf{n}}^i(\boldsymbol{\sigma})), \tag{13}$$

where $\mathrm{v}^i(\boldsymbol{\sigma})$, the *i*th eigenvalue of $\boldsymbol{\sigma}$, $\hat{\mathbf{n}}^i(\boldsymbol{\sigma})$ the corresponding unit eigenvector and

$$\mathbf{m}(\boldsymbol{a}) = \begin{cases} \boldsymbol{a}\boldsymbol{a}^T/|\boldsymbol{a}| & : \boldsymbol{a} \neq 0 \\ 0 & : \boldsymbol{a} = 0 \end{cases}. \tag{14}$$

This allows to compute the tensile and compressive forces exerted on a particle i by one triangle, and compute it for all triangles in the mesh :

$$\boldsymbol{f}_{[i]}^+ = -\frac{area}{2} \sum_{j=1}^{3} \boldsymbol{p}_{[j]} \sum_{k=1}^{3} \sum_{l=1}^{3} \boldsymbol{\beta}_{jl}\boldsymbol{\beta}_{ik}\boldsymbol{\sigma}_{kl}^+ \tag{15}$$

$$\boldsymbol{f}_{[i]}^- = -\frac{area}{2} \sum_{j=1}^{3} \boldsymbol{p}_{[j]} \sum_{k=1}^{3} \sum_{l=1}^{3} \boldsymbol{\beta}_{jl}\boldsymbol{\beta}_{ik}\boldsymbol{\sigma}_{kl}^-. \tag{16}$$

Having the sets of tensile and compressive forces applied by each elements attached to a point, $\{\boldsymbol{f}^+\}$ and $\{\boldsymbol{f}^-\}$, it is possible to calculate the separation tensor :

$$\boldsymbol{\zeta} = \frac{1}{2} \left(-\mathbf{m}(\boldsymbol{f}^+) + \sum_{\boldsymbol{f} \in \boldsymbol{f}^+} \mathbf{m}(\boldsymbol{f}) + \mathbf{m}(\boldsymbol{f}^-) - \sum_{\boldsymbol{f} \in \boldsymbol{f}^-} \mathbf{m}(\boldsymbol{f}) \right). \tag{17}$$

A fracture is happening at the point evaluated if the largest positive eigenvalue v^+ of $\boldsymbol{\zeta}$ is greater than τ, the material toughness. Taking the corresponding eigenvector $\hat{\boldsymbol{n}}^+$, we get the fracture plane's normal.

These calculations are all exactly the same as described in [O'Brien and Hodgins 1999] except for the dimensions of the matrices $\boldsymbol{P}$, $\boldsymbol{V}$ and $\boldsymbol{\beta}$ that are adapted for three nodes instead of four and the number of iterations of the first sum in (15) and (16) that has also been brought down to three.

4.2 Implementation

The methods and algorithms described were implemented in Unity3D with the use of custom scripts. The project's setup is a scene composed of three *GameObject*s; a *Camera*, a *Light* and a *Scene Manager*. To allow interaction with the objects, a basic First Person Camera behaviour was implemented and added to the *Camera object*. The *Scene Manager* object has two *Script Components*. The first one describes the creation of the test scene, with an object that can be fractured and the second one allows for picking and dragging objects in the scene using an external force.

4.3 Interactions

As this project is based of an empty 3D scene in Unity, we added some simple interaction options for the purpose of this project. Basic first person controls were added to the scene's camera.

- $\uparrow$: Move Forward
- $\downarrow$: Move Backward
- $\leftarrow$: Move Left
- $\rightarrow$: Move Right
- $[alt]$ + Click : Rotate

Interaction with the objects is based on a ray cast algorithm slightly modified from [YoungjaeKim 2013]'s. A ray starting from the mouse's position and triggered by a click, tells whether an object was selected or not. If so, an external force is applied on the latter, defined by a vector between the initial click point and the updated mouse's position. Proceeding with a force instead of a simple drag and drop allows for the spring system to be influenced by the displacement of the selected object.

4.3.1 Test Scene Creation

The test scene has only one object which can be fractured and a plane. The breakable object is forced to be a plane itself for simplification purpose, and it's size can be fixed by a parameter available to the user (*MeshPoints*). The decision of limiting the shape of the object to a plane allows to treat it as a simple 2D mesh and implement [O'Brien and Hodgins 1999]'s fracturing algorithm using triangles instead of tetrahedrons.

4.3.2 Fracturable Objects

A fracturable object is made of a set of particles that are managed by Unity's Physics Engine. Those particles are linked together with horizontal, vertical and diagonal springs. The mesh of the object is bound to those particles with each of it's vertices mapped on a corresponding particle. Using the vertex and the index buffers of the mesh, it's possible to define the set of triangles that constitutes it. Since calculations are mostly made triangle-wise, we define a structure that represent one single triangle, referring to the particles that constitutes it. All tensor computations are then made within this structure. The separation tensor is calculated within each particles, using force and stress information sent by all triangles connected to the particle.

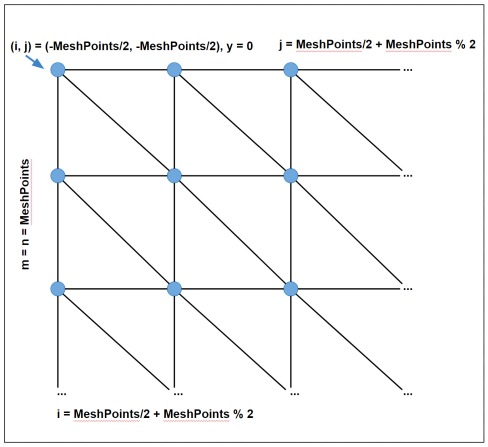

Figure 2: *Structure of the implemented Mesh*

The figure 2 visually present the relation between the different particles, the springs and the triangles. The dimensions of the 2D object are controlled by a single parameter *MeshPoints* that controls how many points on each axis there would be, forming a square shaped plane. The object's particles coordinates vary in x and z between $[-MeshPoints/2, MeshPoints/2 + MeshPoints\%2]$ and the y coordinates are set to 0.

4.3.3 Joints between particles

A challenging part of this project was to define the joints between the particles. three different joints were tested: Spring Joints, Hinge Joints and custom scripted springs.

Spring joints is the most intuitive choice of structure as we are looking for very stiff springs to attach all particles together. After trying a lot of different parameters, it is difficult to get the springs to not only act stiff but to properly transmit force to the linked rigid bodies in our particular setup.

We then tried Hinge Joints. Using this structure along with a spring gave good visual results and a stiffer structure. However, it wasn't satisfying, as this joint impose motion restriction that doesn't apply to our context.

The joints that are used in the final project are springs defined by a custom script. Connecting all particles with these allows for enough deformation within the material to calculate a difference in velocities between two particles that would induce a fracture. The chosen stiffness for all springs is $k = 800$ and the damping is $c = 10$. This happened to be a good compromise between stiffness and stability. Unity's solver gets unstable when the stiffness of the springs is too high and having the parameter k to a even higher value in order to represent tougher materials would make the system blow up. For using those joints, we also added a drag of 5 to the rigid bodies, which were otherwise behaving as frictionless objects.

Unity's physics solver parameters have been set to the maximum number of iterations and the accuracy of the resulting velocity after a bounce was also augmented $Default\ solver\ iteration = 255$ and $Default\ solver\ velocity = 15$. This lowers the overall performance but as we are only testing for one object, the former is not an issue.

4.3.4 Remeshing

The eigenvector returned by the separation tensor corresponds to the normal of the separation plane. Once a fracture is found, we go through all triangles of the mesh to verify if they are connected to the point of fracture and then if they intersect with the separation plane, using [Shilon 2010]'s algorithm. Each points of connected triangles are mapped to a side of another of the plane using a boolean value for which *false* means a node is on the negative side of the plane whereas *true* means the node is on the positive side. We avoid setting the side of a node two times by keeping another boolean value that informs if the calculations have already been made for the particle.

If the plane intersects with a triangle, the latter is separated along the intersection line. First, the particle where the fracture happens is duplicated. Then we test if the plane goes through an already existing node. If so, we undergo simple reassignment of springs between the duplicated node, the original node itself and the existing node on the plane. Otherwise, a new particle is created at the intersection point. This implies a more complex spring reassignment step, as the springs on which the new node is positioned needs to be divided in two and new connections needs to be created to conserve the mesh's integrity. The resulting remeshing is presented in figure 3, where are illustrated the fracture plane, the original and the duplicated particles, the new particle in the first case and the

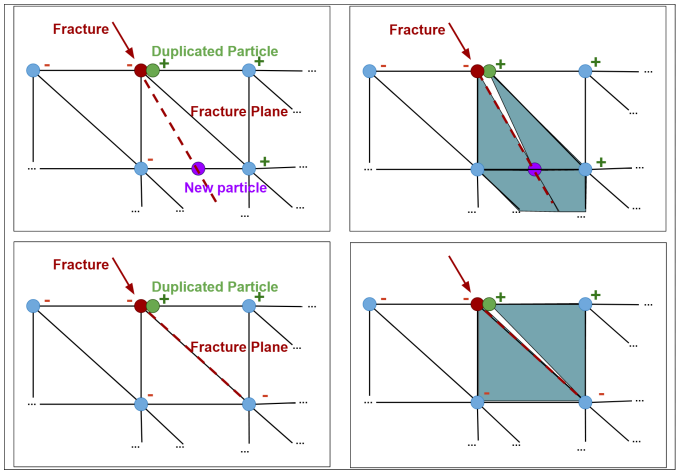

Figure 3: *Remeshing Steps and triangle creation*

sides (plus or minus) of each nodes connecting to the point where the fracture initiates. In turquoise is shown the affected triangles and the new ones created.

Based on the new spring connections, we create new triangles. Using the list of triangles now updated, we can update the index buffer and the mesh consequently.

5 Results

Due to an optimistic prevision of the actual challenge and time needed to implement the remeshing algorithm, the latter was not completed. The challenge lies in the redefinition of the triangles according to the springs connected to a point. A better way of handling the problem at hand would have been to actually define a triangle by three springs and recalculating the particles that forms it's points at each spring modifications. A spring should also have a list of the triangles that are separated by it, such that it is easy to remesh locally an area, knowing what triangles are affected by a modification. Because the code's structure wasn't thought that way initially, this refactoring is quite time exigent and that is why it wasn't integrated in the submitted code.

The attached video shows all other components of the algorithm, including the fracturing calculation in play. This shows the results of the fracturing calculations.

6 Discussion

Even though there is only one *FracMesh* object for the whole scene, we would be able to handle fully divided meshes. Because of the independent handling of the particles, the two sections will physically and visually act as two separate meshes.

While there may be more than one eigenvalues of the separation tensors that are greater than the material's toughness. Our algorithm only handles one separation plane, contrary to what is proposed by [O'Brien and Hodgins 1999].

An accessible improvement to this project would be to extend it to full 3D shapes, like described in [O'Brien and Hodgins 1999]. The fracturing algorithm and calculations are pretty much the same except for the few tweaks made to use three nodes instead of four as described in section 4.1. The remeshing is probably the most challenging part of this extension and would likely account for the biggest amount of work to be done.

In order to manage a larger range of materials, it would be useful to implement a Linear Backward Euler solver in Unity. It would imply getting rid of most of the used features of the *PhysX* Engine and recoding them, inserting the solver calculations into a *Monobehaviour* class' *Update* function. Such an extension to the current project would allow to handle a higher stiffness on the springs and would guarantee much more stability of the system in general.

Finally, the collision detection is only enabled in the project on the rigid bodies. It would be useful to improve the overall project with collision detection on the mesh with itself, to avoid particles to inter-penetrate the mesh.

7 Conclusion

In this project, we first built a simple scene within Unity and implemented basic interaction options. We defined a plane creation sequence that automatically create a fracturable object, constituted of particles, springs and triangles. A mesh is then mapped on top of the particles using their positions as the vertices' coordinates. Once all components are created, we undergo fracture testing at each update of the scene using [O'Brien and Hodgins 1999] algorithm, modified for triangular elements instead of tetrahedral ones. If a point should be fracturing, we calculate the fracture plane's normal and for each triangles connected to the point, we verify if they intersects with the plane. If so, we compute at which point the intersection happens. We also attribute a plane side to each particles connected to the point of fracture which helps proper remeshing. An triangle intersecting the plane is intended to be divided along the intersection line. Springs are consequently redistributed in such a way to take account of the new mesh's shape. The second to last step would be to reform the resulting triangle list. The latter posed a bigger challenge than expected and was therefore not completed. The last step is the general mesh updating which was implemented.

On top of the completion of the triangle redefinition, there would be a lot more improvements that could be made to this project. Amongst others, we could consider implementing our own Backward Euler Solver, be able to handle 3D shapes, add management for many separation planes, etc.

References

O'BRIEN, J. F., AND HODGINS, J. K. 1999. Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press/Addison-Wesley Publishing Co., 137–146.

PARKER, E. G., AND O'BRIEN, J. F. 2009. Real-time deformation and fracture in a game environment. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 156–166.

RIVERS, A. R., AND JAMES, D. L. 2007. Fastlsm: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26*, 3 (July).

SHILON, O., 2010. Determining the intersection of a triangle and a plane. StackOverflow Answer. Accessed: 2017-04-30.

YOUNGJAEKIM, 2013. Drag drop game objects (without rigidbody) with the mouse. Unity3D Forum Answer. Accessed: 2017-04-30.