

Master's Project

*Generating Crack Patterns with Hybrid  
Physical/Heuristic Approach*

Albert Jian Sheng Tan-Mulligan

MSc Computer Animation and Visual Effects

NCCA

<https://github.com/AlbertTM8/>

## ***Abstract***

Crack patterns are important for creating realistic objects that have experience wear and tear. It is seen in many real-life objects such as ceramics, glass, mud, and concrete etc. As such, it is an important part of creating realism in digital scenes as without them, scenes seem too perfect. Many methods exist for the creation of such patterns. They are mainly split into physically based and non-physically based, however approaches have been proposed that combine physically based, real-world methods with artistic/heuristic methods. A method is discussed in this paper that takes one of those methods and alters it slightly to improve artistic understanding and improve processing time.

# ***Acknowledgements***

I would like to thank my tutors, Jon Macey and Jian Chang, for the advisement and mentorship throughout this project.

I would also like to thank Jon Macey, Jian Chang, Jian Zhang, Ian Stephenson, Phil Spicer, Veno Prendergast and LiHua You for being wonderful teachers throughout this academic year.

# Contents

## **1. Introduction**

## **2. Background**

*2.1 Physically Based Approaches*

*2.2 Non-Physical Methods*

*2.3 Hybrid Methods*

## **3. Methodology**

### **3.1 Stress Field**

*3.1.1 Weight Field*

*3.1.2 Radial and Concentric Force Vectors*

*3.1.3 Converting to 2D Coordinates*

*3.1.4 Stress Tensors*

### **3.2 Houdini Implementation**

*3.2.1 Weights*

*3.2.2. Radial and Concentric Force Vectors*

*3.2.3 Converting to 2D Coordinates*

### **3.3 Forces at Nodes**

*3.3.1 Barycentric Matrix*

*3.3.2 Force Equation*

### **3.4 Houdini Implementation**

*3.4.1 Stress Tensor Eigen Decomposition*

*3.4.2 Barycentric Basis Matrix and Testing*

*3.4.3 Node Forces*

*3.4.4 Separation Tensor*

## **4. Post-Processing/Visualisation**

## **5. Analysis**

## **6. Conclusion and Future Work**

## **1. Introduction**

In the computer graphics field, realistic and feasible crack patterns can add realism to a scene. While most methods for doing so take either a physical or non-physical approach, I present an alteration to an already existing method presented by Iben et al. This original method creates identifiably realistic crack patterns, while also affording high levels of heuristically based control for artists to use. It can create a variety of crack patterns, specifically focused on the brittle fracture of ceramics and glass, on the surface of 3D meshes, as opposed to fully fracturing a volumetric mesh in its entirety. My deviation from this algorithm takes away some physically based realism of the method, however it reduces the run time of the algorithm by taking a few shortcuts. It also adds extra parameters to give artists more control. While the original method creates cracks of varying scenarios, we focus on cracks developed through impacts in this paper. Some other crack patterns are presented but not in detail.

The method takes a triangular mesh and develops a stress field through an intuitive and artistic approach. From this stress field, we take a physically based method loosely based on Finite Element Modeling to decide the placement, direction, length and shape of cracks. We then can define several parameters that will give artistic control to an artist to control the way the crack is developed. The method does not produce the most realistic cracks, being heavily based on heuristics after all, however it produces very quickly identifiable and plausible crack patterns. Future works are then presented.

## **2. Background and Past Works**

Various methods of generating crack patterns exist within the computer graphics community. Generally, they are split into two different types, physically based and non-physically based. Physically based approaches for developing crack patterns, mostly comprised of finite element models, aim to accurately represent the mechanical behavior of materials and the propagation of cracks based on underlying physical principles. Non-physical approaches to generating crack patterns refer to techniques that form and propagate cracks in materials without strictly following physical laws and principles. They prioritize artistic or visual outcomes over physical accuracy. While not being the most realistic, as they do not typically adhere to physics, these approaches can be convincing and aesthetically pleasing crack patterns.

### ***2.1 Physically Based Approaches***

Within the Computer Graphics field, a few notable physically based approaches for creating crack/fracture patterns include:

- Terzopoulos et al. created early work in computer graphics with methods for simulating elastic and inelastic deformation. (Terzopoulos and Fleischer, 1988) They introduced these concepts to the graphics community.
- Mass-spring systems were used by Muguercia et al. to simulate fracture and cracks, which were simple but had shortcomings when it came to shear resistance and calculating fracture location and direction. (Fumoto, 2022) (Skjeltorp and Meakin, 1988)
- Finite Element models(FEM) are used extensively to simulate ductile/brittle/interactive fracture. Specifically for generating cracks, rather than object breaking and destruction, Alshoaibi and Fageehi used FEM to model crack growth in 2D structures. Swenson and Ingraffea defined a new

way of thinking which modeled dynamic fracture that allowed for crack propagation along non-predefined mesh line through remeshing.

Physically-based techniques obviously offer more accurate results than non-physical. If a simulation accounts and acts according to physical laws, as a real-world object would, then the realism is very high. The problem with physically based is that they are computationally complex and don't offer the best interpretations for artists to use them. They might need to discretize the geometry to infinitesimally small pieces and iterate through equally small-time steps causing long simulation times. Given a large number of nodes in a mesh discretization, the input data can be very large as well. (caeassistant.com, 2022)

## 2.2 Non-Physical Methods

Here are a few notable examples of non-physical crack pattern generation techniques:

- Procedural crack patterns:
  - Voronoi Diagrams - Voronoi diagrams are surprisingly simple fundamental concept that is used to describe the structure of "polycrystalline alloys, cellular foams, geomaterials, trabecular bone, and other materials that exhibit cell-like features." (Sukumar and Bolander, n.d.) Essentially, Voronoi diagrams split a space into regions based on the proximity to a given set of points, scattered through the space. This tessellation process creates regions of all the space closer to one point than any other point.
  - L-Systems – L-Systems or "Lindemayer" systems can be defined as "a formal language which is able to define growing processes of structures by different grammars". (Megumi Takato and Yohei Nishidate, 2018) They are commonly used within Houdini to create natural shapes such

as trees and bushes. An example of L-systems being used to create crack shapes can be seen below. (Megumi Takato and Yohei Nishidate, 2018)



Figure 1: L-System Patterns

- **Interactive Techniques** - Interactive approaches allow artists to directly manipulate crack patterns in real time using digital tools. Artists can paint or sculpt cracks onto a digital surface, adjust parameters, and see immediate visual feedback. The patterns are then post processed to achieve the desired result. (Martinet et al., 2004) Uses this technique to generate 2D crack patterns, and then build a “geometric skeleton” which directs the carving of a volume, this creating crack depth and width.
- **Pattern Synthesis** - Pattern synthesis methods integrate existing crack patterns to create new and interesting ones through editing. One such example is (Hsieh et al., 2004) that uses image processing operations to take 2D images and project them onto 3D objects.

The problems with non-physical based methods are that they inherently have a lack of physical accuracy. Due to this, they may have problems predicting crack behavior, producing artificial and homogeneous results. They may also require extra setup so that cracks are oriented correctly, such as a technique called pre-fracturing where objects are cut into different pieces by artists, or procedurally split beforehand to decide crack placement and direction. (Müller, Chentanez and Kim, 2013)



Pre-fracturing is a popular approach used in gaming and film because of the controllability it gives to artists. Non-physically based methods often apply simple algorithms or procedural techniques, such as Voronoi patterns and other geometry-based methods. As a result, changing in material properties or force/stress interactions will not be reflected in the fractures as they are not originally considered in the technique. (Saty Raghavachary, 2002) The list is quite long, but mainly it all boils down to the fact that real-world processes and mathematical representations are not typically considered in these methods.

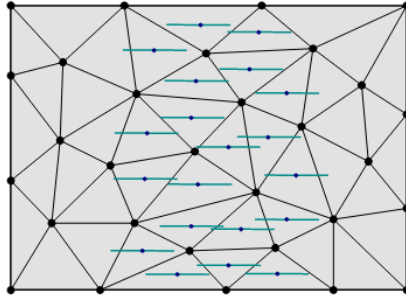
### **2.3 Hybrid Methods**

A couple hybrid methods that employ both physical and non-physical are Iben et al and Valette et al. (Iben and O'Brien, 2006) (Valette, Stéphanie Prévost and Lucas, 2006) For the work of Valette et al, they focused on creating cracks caused by shrinking 3D materials. Their work starts with creating both a stress field and a 2D "precalculated path" for the crack, essentially just procedural pre-fracturing. The stress field is defined with the thickness of the material, formed into a 2D mesh of cracks and then interpolated to a parametrized 3D mesh. Using their 3D "cellular automata" model, they calculate crack widths that are applied to the interpolated cracks.

My method is largely based on the work of Iben et al. in "Generating Surface Crack Patterns". I am going to regularly refer to the work done in this paper as "the original method". The methodology section goes into far more detail; however, the algorithm can be generalized by this pseudo-code:

- (1) Initialize the stress field according to heuristics and optionally evolve it with relaxation.
- (2) Compute the failure criteria for each node and store the nodes in a priority queue based on this value.
- (3) While failure can occur and the user wishes to continue:
  - (a) Crack the mesh at the node associated with the top of the priority queue.
  - (b) Evolve the stress field:
    - (i) Perform relaxation.
    - (ii) Optional: Add shrinkage tension and/or curvature biasing.
  - (c) Update the mesh information and priority queue.
- (4) For display, either:
  - (a) Post-process the mesh by moving the vertices to give the cracks width and filling in the gaps with side-walls for the cracks.
  - (b) Directly render crack edges.

First, a stress field of tensors, as illustrated below, is developed through an artistic approach across a triangular mesh. This stress field is then analyzed, and the crack location and direction are developed based on the largest tensor value. Then a while loop starts that only stops when there are no more nodes that meet the failure criteria, or the user wishes to stop. In each loop, the node with the highest priority (so the node with the most stress) is cracked in a direction based on the stress tensor. This action then updates the values of the stress tensor at that node and the newly generated crack tip causing them to be earlier in the priority queue, and therefore come up more often, causing long crack lines. The triangle that it cracks "into" is then remeshed to show this crack along its plane. Relaxation is essentially a diffusion process that dissipates the stress tensor along the surface of the mesh. It is important for realism as it is used for analyzing static stress conditions. (apps.dtic.mil, n.d.)

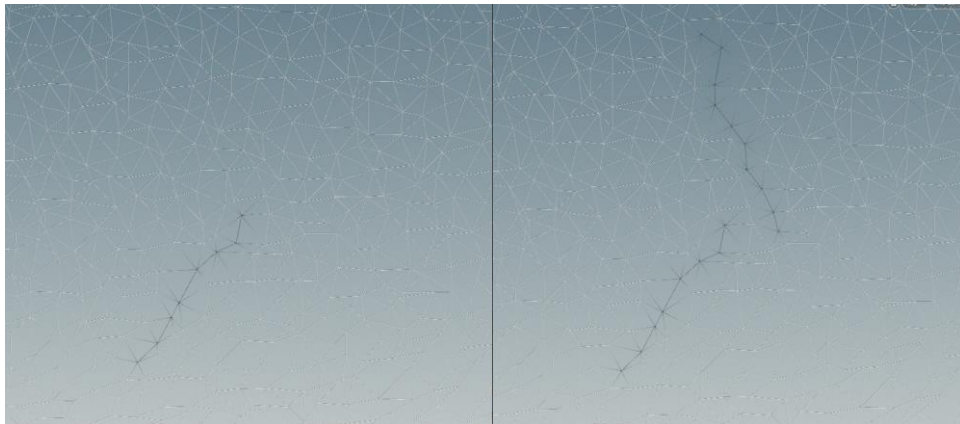


*Figure 2: Depiction of Stress Field (Iben and O'Brien, 2006)*

Stress analysis typically generates more physically correct fracture behavior but has less controllability and speed. (Matthias Müller and Gross, 2004) However, using the mixed method retains that controllability and speed by not calculating stresses based on physical interaction between objects. The heuristic initial step also helps artists to understand how to create target crack shapes better. Using hybrid methods typically produces a result that is still not technically correct but can be more physically feasible than a purely non-physical method. The goal of this paper is to create crack shapes/patterns that are computationally efficient, identifiable and realistic.

### 3. Methodology

My algorithm follows the methodology in the pseudo-code above for the first two steps. For step 3, rather than crack each node iteratively, and having the algorithm eventually come back to propagate the cracks from previous loops, I chose to have one crack “length” form from beginning to end. This will make more sense in the following sections. From then I visualize the crack by setting poly-wires from each node along the direction that it cracked. As written in the original paper, their methodology focuses on the generation of cracks in brittle materials like ceramics and glass. They also depict a heuristic method for the first step, which focuses on impact cracks. This paper will focus on the generation of semi-realistic, believable impact cracks. To that end, I set a few success benchmarks for the project based on real life impact cracks on glass, that will be used in the discussion section to analyze the result of my method.



*Figure 3: Crack lengths develop to their full length before a new one begins development*

- Stress impacts cause two types of fractures, radial and concentric (also known as spiral) fractures. Radial cracks form outward from the impact point in a line and concentric cracks form in a circular shape emanating from the impact point. (Tiwari et al., 2019) Therefore, the

presence of predominantly radial and then some connecting concentric lines was deemed as a success criterion.

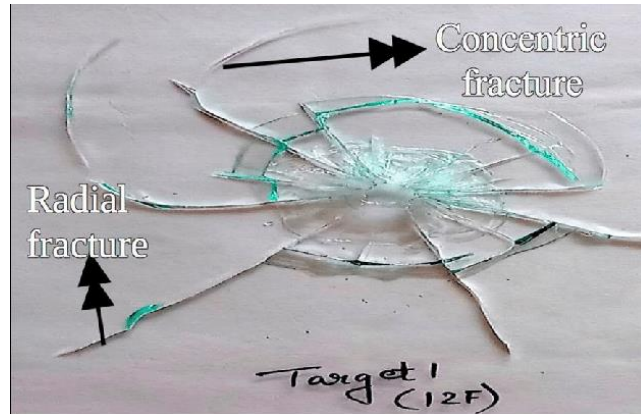


Figure 4: Diagram of Radial and Concentric Cracks (Tiwari et al., 2019)

- In mechanical and impact cracks, crack lines tend to form as meandering single/branching lines. Therefore, having many separate small and jagged lines, or crooked lines would be unrealistic, and this was also deemed as a success criteria. (Anon, n.d.)

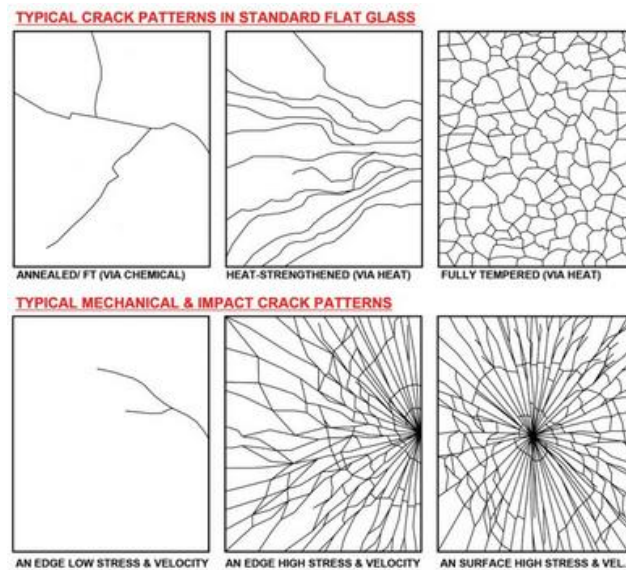


Figure 5: Meandering and branching lines (Top) and Radial and Concentric lines (Bottom). (Anon, n.d.)

- Obviously, having the cracks be primarily localized around the impact point, and then sparser further away from the origin point.

I chose to use the Software program Houdini to develop this as its procedural nature and vex coding language gave a very good amount of control on the mesh. This paper assumes that the reader has a decent understanding of the Houdini suite. A few things to consider:

- Houdini uses a Y-up right-handed coordinate system. This may be important as vectors are in a  $(x, y, z)$  format, which translates to matrices.
- Attribute wrangle nodes cannot compute on the output of itself. While this may seem self-explanatory, this means that even if the input geometry is changed within the code, the next lines of cannot interact with, for example, the new point that was created. Also, the code is not looped once per code, but rather run per line, per point/vertex/primitive (based on input). This may also affect logic in certain scenarios, so keep this in mind when reading code.

Each section will be comprised of a technical section to discuss the mathematics and theory of the method, followed by a practical implementation section to discuss how this was implemented within Houdini.

### **3.1 Stress field**

To start off with a triangle mesh is used, and for simplicity I developed my proof-of-concept using a simple grid that was remeshed into triangles. The first thing to do is calculate the stress field through a heuristic/artistic method. The stress field can be determined using any random shape, for example this

crack shape below was developed using the shape of the word NCCA. For this specific example, we will be developing the stress field with the intension of creating impact cracks.

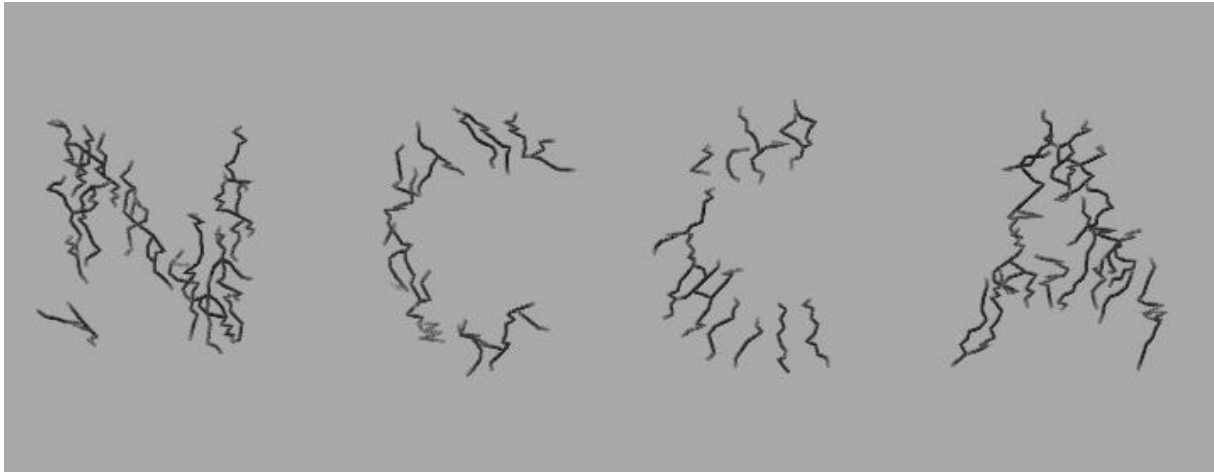


Figure 6: NCCA Crack Shape

### 3.1.1 Weight Function

To determine the stress function, the first step is to determine radial and concentric forces, which influence the shape and values of the stress field. To aid artists with understanding the input of the simulation, an intuitive set of inputs is used to create a weight that will be the scaling factor of the radial and concentric forces. This weight variable eventually determines the shape and values of the stress field. While the values across the profile of the weight variable can come in any pattern, to model impact shapes this was modeled using the equation:

$$w_i = 2.0 \cos \left( \frac{d_i}{r} \right),$$

*Equation 1*

where the  $W_i$  is the weight of primitive number  $i$ ,  $r$  the radius of the impact (input parameter from the user), and  $d_i$  the distance from the centroid of the triangle to the point of the impact (also a user specified input:  $P$ ). The outcome of this equation returns a circular weight profile that has high values in the center and falls off towards the edges of this circle with radius.

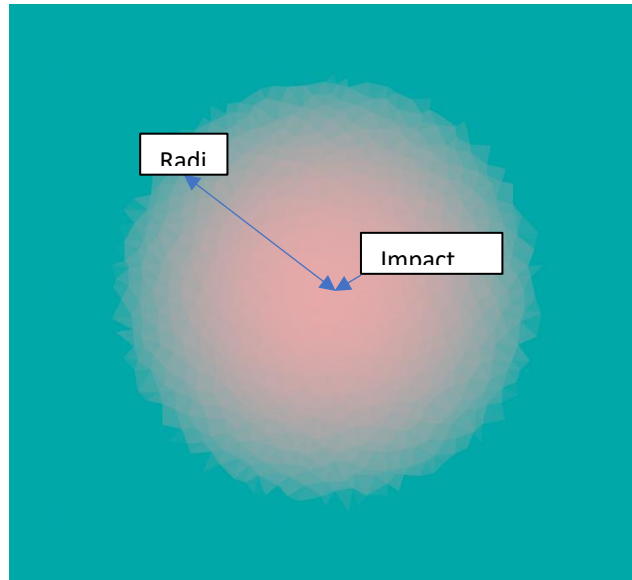


Figure 7: The weight profile, red representing high scaling factor.

### 3.1.2 Radial and Concentric Force Vectors

The radial vector direction for each primitive is calculated by:

$$\left( \frac{\mathbf{c}_{[i]} - \mathbf{p}}{|\mathbf{c}_{[i]} - \mathbf{p}|} \right)$$

Equation 2

where  $c_{[i]}$  is the center of the primitives and  $p$  the impact point. Multiplying this by the weight we get the full radial vector with which to construct our stress field with.



$$\mathbf{v}_{rad} = W_i \left( \frac{\mathbf{c}_{[i]} - \mathbf{p}}{|\mathbf{c}_{[i]} - \mathbf{p}|} \right)$$

Equation 3

Cracks form in a radial direction first, and then form in a concentric fashion, as discussed before.

Therefore, the forces need to be balanced in the correct direction so that the radial cracks form before the concentric ones. Since cracks form in a perpendicular direction to the largest forces, the concentric forces need to be larger. Therefore, the concentric vectors are multiplied by a factor of 1.25 so that the eventual stress field (and by extension the node forces) are larger in the concentric direction, perpendicular to the direction of the impact point. Therefore the concentric vector equation is:

$$\mathbf{v}_{circ} = 1.25(\mathbf{v}'_{rad})^{\perp}.$$

Equation 4

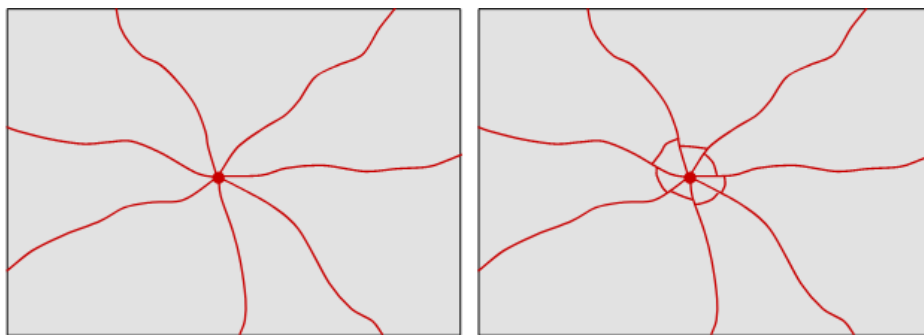


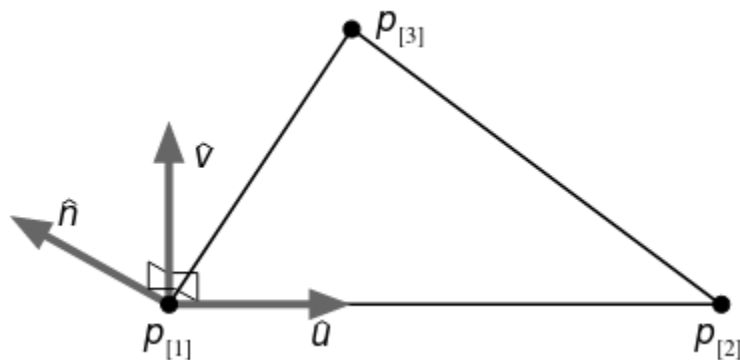
Figure 8: Cracks develop in a radial direction before concentric (Iben and O'Brien, 2006)

### 3.1.3 Converting to 2D Coordinates

In order to simplify the calculations, and simplify the calculation of eigenvectors later, the radial and concentric force vectors were converted to a 2D space determined by the edges of the triangles.

Defining the 3 nodes of a triangle as  $P[1]$ ,  $P[2]$  and  $P[3]$ , a new coordinate system determined by the axis  $\mathbf{u} = P[2] - P[1]$ , as illustrated in the figure below. The axis  $\mathbf{n}$  is determined by  $\mathbf{n} = \mathbf{u} \times (P[3] - P[1])$ , and by extension  $\mathbf{v} = \mathbf{n} \times \mathbf{u}$ , therefore defining the new coordinate system with  $\mathbf{u}$  and  $\mathbf{v}$ .

However, this is only fully needed when the mesh is of a 3D nature, however the grid that is used in this implementation is already 2D. Therefore to convert my forces into the local coordinates of the I simply rotated the forces along the tangential axis (tangential to the grid) by  $\theta$ , where  $\theta$  is the angle from the x-axis to  $\mathbf{u}$ .



*Figure 9: The new 2D coordinate system of each triangle. (Iben and O'Brien, 2006)*

### 3.1.4 Stress tensors

The stress tensor describes the forces acting on infinitesimal planes within the material, but in this case, infinitesimal means the size of a primitive. The stress tensor for each primitive makes up the stress field. A stress tensor expresses how much force a section of the mesh (the primitive) acts on a node. Below is

an example visualization of a hypothetical stress field, again with concentric stresses being larger than radial stresses.

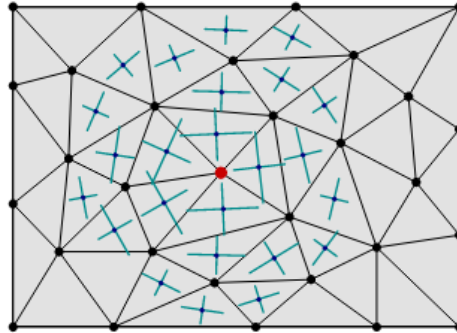


Figure 10: Stress field visualization. (Iben and O'Brien, 2006)

To convert the radial and concentric forces into the stress tensor, we multiply the now 2D vector by the transpose of itself (i.e. 2x1 by 1x2 multiplication) to form a 2x2 tensor, as in the equation below. Doing so with both radial and concentric force vectors develops the entire initial stress field to be formed, represented by  $\sigma$ .

$$\sigma' = \sigma + \mathbf{V}_d^T \mathbf{V}_d.$$

Equation 5: Stress Tensor Update

## 3.2 Houdini implementation

I created a grid in Houdini, applied a point jitter for random variation to the shape. The random variation in the shape, size and orientation is important for creating high stress points in the discretization process. I then applied the remesh node to split the quads into tris.

### 3.2.1 Weights

For the initial weight field, I used a primitive attribute wrangle. First, I calculated the centroid of each triangle and saved it as attribute `@center`. To find the point numbers of each node on a primitive, I used the function `primpoints()`, and then used `getattrib()` to find their position attribute `P[i]`. To simplify things, (0,0,0) was my impact point. From there, I simply used equation 1, with radius = 3, to find the weights and set it as `@weight` attribute per primitive. The code can be seen below.

```
VExpression
1 int points[] = primpoints(0, @primnum);
2 //primitive points
3 vector @center = {0.0, 0.0, 0.0};
4 vector p0 = getattrib(0, "point", "P", points[0],0);
5 vector p1 = getattrib(0, "point", "P", points[1],0);
6 vector p2 = getattrib(0, "point", "P", points[2],0);
7 //centroid
8 @center[0] = (p0.x+p1.x+p2.x)/3;
9 @center[1] = (p0.y+p1.y+p2.y)/3;
10 @center[2] = (p0.z+p1.z+p2.z)/3;
11
12 float dx = @center[0];
13 float dy = @center[1];
14 float dz = @center[2];
15 float @distance = 0;
16
17 //distance from center
18 @distance = sqrt(dx * dx + dy * dy + dz * dz);
19 float @weight = 0.0;
20 |
21 @weight = 2*cos(@distance/3);
22 if(@weight<0){
23 @weight=0;
24 }
25
```

Figure 11: Code for Weights

### 3.2.2 Radial and Concentric Force Vectors

Using another primitive attribute wrangle node, I first calculated the radial force vectors. Following equation 2, I simply plugged in the values of the centroid ( $P = 0, 0, 0$ ) and the weights to create attribute `@radial`. To convert the vector to the 2D coordinate system, I find the angle between *u and the x axis (in this case v)*. Then, if the z axis of vector *u* is above 0, then I invert  $\theta$  to  $-\theta$ , so that the angle is either clockwise for negative values and counter-clockwise for positive values

I initialized a transformMatrix as an identity matrix. I then rotated it by the calculated angle  $\theta$  around the y-axis to get it to be in the right coordinate system. Using the @radial attribute, I then apply the transformation matrix to it. Another vector @concentric is defined by scaling @radial by a factor of 1.25. The transformMatrix is reset to an identity matrix and is then rotated by 90 degrees (1.5708 radians) around the y-axis. The concentric vector is then transformed by this new matrix so that it is perpendicular to the radial vector.

```
VExpression
1 //radial vector
2 vector radial = {0.0,0.0,0.0};
3 radial = @weight*(@center)/abs(@center);
4
5 //find the transformation to the correct coordinates
6 int points[] = primpoints(0, @primnum);
7 vector p0 = getattrib(0, "point", "P", points[0],0);
8 vector p1 = getattrib(0, "point", "P", points[1],0);
9 vector p2 = getattrib(0, "point", "P", points[2],0);
10 vector newx = p2 - p0;
11 normalize(newx);
12 vector ex = {1,0,0};
13 matrix3 transformMatrix = ident();
14 float angle = acos( dot(ex,newx) / (length(ex)*length(newx)));
15 angle = -angle;
16 rotate(transformMatrix, angle, {0, 1, 0});
17
18 // Apply the transformation matrix to convert the vector
19 vector @radialt;
20 v@radialt = transformMatrix*radial;
21
22 // Scale Concentric Direction and rotate 90 degrees
23 vector @concentric;
24 v@concentric = 1.25*(@radialt);
25 transformMatrix = ident();
26 rotate(transformMatrix, 1.5708, {0, 1, 0});
27 v@concentric = transformMatrix*v@concentric;
28
```

Figure 12: Code For Radial and Concentric Vectors

### 3.2.3 Converting to 2D

At this point I still was using 3x3 matrices and vectors, just with the y value as 0 because the rotate function in Houdini doesn't support non-3-dimensional arrays. To convert the @radial to a 2-dimensional I simply assigned a new vector2 as (@radial.x , @radial.y). A matrix2 attribute

@stress\_tensor is declared. I then calculated a matrix2 by multiplying the 2x1 @radial vector by the 1x2 transpose of itself and added it to the @stress\_tensor attribute. I then did the same for the @concentric vector and added it to the @stress\_tensor attribute.

```
VExpression
1 //radial stress tensor update
2 vector2 original = set(v@radial.x, v@radial.z);
3 matrix2 @stress_tensor;
4 matrix2 update = set(original.x*original.x,
5 original.x * original.y,
6 original.y * original.x,
7 original.y * original.y
8 );
9 2@stress_tensor = 2@stress_tensor + update;
10
11 //concentric stress tensor update
12 original = set(v@concentric.x, v@concentric.z);
13 update = set(original.x*original.x,
14 original.x * original.y,
15 original.y * original.x,
16 original.y * original.y
17 );
18
19 2@stress_tensor = 2@stress_tensor + update;
20
```

Figure 13: Code for Stress Tensor Calculations

### 3.3 Forces at the Nodes

We want to know how stress affects the nodes. We need to calculate the forces applied on a node by each connected primitives, and then sum them up to find the total force applied to a node.

#### 3.3.1 Stress Tensor Eigen Decomposition

To calculate the positive and negatives stresses, eigenvalues and corresponding eigenvectors of the stress tensor are calculated. In doing do, we find “the principal stresses (the eigenvalues of  $\sigma$ ), the maximum and minimum normal stresses acting upon the principal planes of stress.” In 2d, there are only 2 principal planes, which are orthogonal to each other and contain only normal stress. Therefore, by

finding the positive and negative eigenvalues, we decompose the stress tensor into the positive and negative stresses. To find both positive and negative components, the equation is therefore:

$$\begin{aligned}\boldsymbol{\sigma}^+ &= \sum_{j=1}^2 \max(0, v^j(\boldsymbol{\sigma})) \mathbf{m}(\hat{\mathbf{n}}^j(\boldsymbol{\sigma})) \\ \boldsymbol{\sigma}^- &= \sum_{j=1}^2 \min(0, v^j(\boldsymbol{\sigma})) \mathbf{m}(\hat{\mathbf{n}}^j(\boldsymbol{\sigma})) .\end{aligned}$$

*Equation 6: Positive and Negative Stress*

where  $v^i(\boldsymbol{\sigma})$  corresponds to the  $i^{\text{th}}$  eigenvalue ( $i \in \{1,2\}$ ),  $\hat{\mathbf{n}}^i(\boldsymbol{\sigma})$  corresponds to the  $i^{\text{th}}$  eigenvector and  $\mathbf{m}$  is a function computing the outer product of a vector divided by its length.

Therefore, the negative and positive eigenvalues are sorted through the min/max functions, and then multiplied by the normalized tensor product of their corresponding eigenvector to reproduce a 2x2 tensor value. This is then done for each eigenvalue/eigenvector pair and then positive and negative values are summed separately. These positive and negative values are then the values used in the force function described in section 3.3.3.

### 3.3.2 Barycentric Matrix

Barycentric coordinates are a set of coordinates used to represent points within a simplex by assigning weights to the vertices. More simply put, they represent the coordinates of a point within the triangle relative to the vertices. (Weisstein, n.d.)

For example, the barycentric coordinates for the centroid of a triangle would be (0.33, 0.33, 0.33), and for point 1 would be (1,0,0). Barycentric coordinates are useful to interpolate data covering a triangle's surface, but in our case, we are doing the inverse where we want to interpolate the value at the vertices based on the values of the triangle. A barycentric basis matrix was derived by Iben et al. to convert a

position in triangle coordinates to barycentric coordinates, which will be relevant in the force calculation:

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

Equation 7: Barycentric Basis Function

M[1], M[2], M[3] in this case are the node positions in the triangle's coordinate system.

### 3.3.3 Force Equation

Using the barycentric matrix, a force equation for which the derivation can be found in O'Brien and Hodgins, 1999 is as shown below. This force<sub>[i]</sub> is the amount of that force that an element/primitive exerts on node i:

**Let  $\mathbf{v}_{[i]}$  denote  $\mathbf{v}$  at node  $i$  and  $\mathbf{v}_i$  denote element  $i$  of  $\mathbf{v}$ .**

$$\mathbf{f}_{[i]} = -A \sum_{j=1}^3 \mathbf{p}_{[j]} \sum_{k=1}^2 \sum_{l=1}^2 \beta_{jl} \beta_{ik} \sigma_{kl},$$

Equation 8: Force Calculation per node, per element.

In the equation, A is the area of the primitive,  $\beta$  is the barycentric basis matrix as defined in the section above, p is the node world coordinates and  $\sigma$  is the stress tensor. Just to be clear on notation, f[0] is f at node 0, and  $\sigma_{22}$  is the bottom-right element in the 2x2 stress tensor. To find the total force on a node, we simply sum the forces acting on that node by the primitives it is connected to. For our methodology, we split the stress tensor into positive and negative as described in section 2.1. The positive and negative stress tensors  $\sigma^+$  and  $\sigma^-$  are then substituted into the equation above separately.



## 3.4 Houdini Implementation

### 3.4.1 Stress Tensor Eigen decomposition

First, to find the eigenvalues and eigenvectors I used to methodology described in the Harvard.edu (2023), as per usual within a primitive attribute node. In code this is a simply mathematical assignment followed by some if-else statements. Since our stress tensor is a real and symmetrical 2x2 matrices, the calculations are relatively cheap. Once eigenvectors and eigenvalues are calculated, they are eigenvectors are normalized. (Note: Using the notation according to the original method, if both c and b values of the 2x2 tensor are non-zero, then it does not matter which calculation is done because once they are normalized the values will be the same). The function Mfunc takes a 2D vector input and returns a 2x2 matrix based on outer product and normalization. This function will be used many times in the rest of our methodology.

Positive and negative stress components (@posstress, @negstress) are calculated using the eigenvalues and eigenvectors as described in the previous section.

```
VExpression
1 matrix2 m = 2@stress_tensor;
2
3 // Calculate eigenvalues
4 float trace = getcomp(m,0,0) + getcomp(m,1,1);
5 float determinant = getcomp(m,0,0) * getcomp(m,1,1) - getcomp(m,0,1) * getcomp(m,1,0);
6 float eigenvalue1 = (trace/2)+sqrt(((trace*trace)/4)-determinant);
7 float eigenvalue2 = (trace/2)-sqrt(((trace*trace)/4)-determinant);
8
9
10 // Calculate eigenvectors
11 vector2 eigenvector1;
12 vector2 eigenvector2;
13 if(getcomp(m,1,0)!= 0 ){
14     eigenvector1 = set(eigenvalue1-getcomp(m,1,1), getcomp(m,1,0));
15     eigenvector2 = set(eigenvalue2-getcomp(m,1,1), getcomp(m,1,0));
16 }
17 else if(getcomp(m, 0, 1)!= 0){
18     eigenvector1 = set(getcomp(m,0,1), eigenvalue1-getcomp(m,0,0));
19     eigenvector2 = set(getcomp(m,0,1), eigenvalue2-getcomp(m,0,0));
20 }
21 else{
22     eigenvector1 = set(1,0);
23     eigenvector1 = set(0,1);
24 }
25 eigenvector1 = normalize(eigenvector1);
26 eigenvector2 = normalize(eigenvector2);
27
```

Figure 14: Eigenvalues and vectors calculation

```
VExpression
28
29 matrix2 @posstress;
30 matrix2 @negstress;
31 matrix2 Mfunc(vector2 input){
32 if(input == 0){
33 return 0;
34 }
35 else{
36     matrix2 ret = outerproduct(input, input);
37     ret /= length(input);
38     return ret;
39 }
40 }
41
42 float max = max(0, eigenvalue1);
43 matrix2 m1 = Mfunc(eigenvector1);
44 m1 *= max;
45 @posstress += m1;
46
47 max = max(0, eigenvalue2);
48 m1 = Mfunc(eigenvector2);
49 m1 *= max;
50 @posstress += m1;
51
52
```

Figure 15: Eigenvalues and vectors calculation

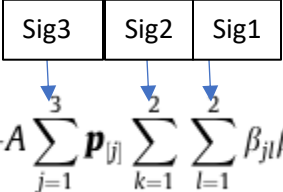
### 3.4.2 Barycentric Basis Matrix and Testing

To calculate the barycentric matrix, I retrieved the positions ("P") of the three points on each triangle (p0, p1, and p2) using the getattrib function, I converted the P position values to the triangle's coordinate system by translating and rotating them accordingly. New 2D vectors m1, m2, and m3 are created by removing the y component of p0,p1, p2. (m1, m2 and m3 define the barycentric basis matrix). I then constructed the matrix basis matrix according to equation 7 and used the build-tin invert function to invert the matrix. This attribute is stored in attribute @bas as a 3x3 matrix.

Separately, in order to test the function, I created a 3x1 matrix m, that I input a separate quad split into two triangles of the same size. As discussed before, if the barycentric basis matrix is multiplied by the vertex points in local triangle coordinates, the outcome should be (1,0,0) for point 1, (0,1,0) for point 2 etc. By manually changing the values of matrix m, I could then multiply the basis matrix @bas × m, and check that the outcome of this multiplication was as expected, testing the validity of the barycentric basis function.

### 3.5 Node Forces

I first calculated the area of the triangle formed by these three points using the formula: Several variables are initialized to store cumulative values for calculating forces (sig = positive and sin = negative). As shown below sig1 is the accumulation of the first summation (Sigma 1, or the right-most sigma), and the same applies to sig2/3.

$$f_{[i]} = -A \sum_{j=1}^3 p_{[j]} \sum_{k=1}^2 \sum_{l=1}^2 \beta_{jl} \beta_{ik} \sigma_{kl}$$


Nested loops iterate over indices as described in the function to calculate the forces based on the basis matrix @bas and stress tensors(@posstress and @negstress). The getcomp() function returns the value of the input matrix (getcomp(3@bas, i, k) = @bas[i][k]). The calculated forces (forces\_pos and forces\_neg) are stored in arrays, since later I will need to positive and negative force for each primitive on a node. These arrays are then stored as point attributes for the corresponding point using the setpointattrib() function.

```

for(int i = 0; i < 3 ; i++){
for (int j= 0; j<3; j++){
  for(int k = 0; k<2; k++){
    for(int l = 0; l<2; l++){
      sig1 += getcomp(3@bas, j, l)*getcomp(3@bas, i, k)*getcomp(2@posstress, k, l);
      sin1 += getcomp(3@bas, j, l)*getcomp(3@bas, i, k)*getcomp(2@negstress, k, l);
    }
    sig2 += sig1;
    sig1 = 0;
    sin2 += sin1;
    sin1 = 0;
  }
  sig3 += ps[j]*sig2;
  sig2 = 0;
  sin3 += ps[j]*sin2;
  sin2 = 0;
}
forces_pos[i] = -area*sig3;
forces_neg[i] = -area*sin3;
setpointattrib(0, "Cd", points[i], forces_pos[i], "add");
setpointattrib(0, "Cd", points[i], forces_neg[i], "add");
setpointattrib(0, "posf", points[i], -area*sig3, "append");
setpointattrib(0, "negf", points[i], -area*sin3, "append");
}

```

Figure 16: Code For Calculating Forces

### 3.5 Separation Tensor

At this point, per node, we have the forces acting on it from each primitive calculated. Therefore, we need to organize it in a way that can decide where the nodes should be cracking. To do so, we balance negative and positive forces exerted on a node, using a “separation tensor” presented in O’Brien and Hodgins (1999), which is essentially just a variation on the classic stress tensor. Since we’re now using 3-dimensional coordinates, this separation tensor is of a 3x3 quantity. We use our 3D forces to calculate this 3x3 matrix using the following formula.

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

Equation 9: The Separation Tensor

Where  $\mathbf{f}^+$  and  $\mathbf{f}^-$  are the total positive forces acting on a node, and  $\mathbf{f} \subseteq \mathbf{f}^+$  corresponds to all the positive forces acting on a node from each connected primitive. Again, the  $\mathbf{m}$  function calculates the normalized

outer product of the input vector. This node is used to decide where the node cracks using its eigen decomposition. The material fails at a node when the largest eigenvalue is greater than the material toughness, in the direction of the corresponding eigenvector.

### 3.6 Implementation

Unfortunately, the methodology used before for calculating eigen decompositions doesn't apply for 3x3 matrices. As a result, I used the help of the numpy library in python, only for this section. Using a python node in Houdini, I implemented the following code using Houdini's python API.

```
Python Code
1 import hou
2 node = hou.pwd()
3 geo = node.geometry()
4 import numpy as np
5 i = 0
6 for point in geo.points():
7
8     A = np.array(point.attribValue("separation"))
9     A = np.array([[A[0], 0, A[2]],
10                  [0, 0, 0],
11                  [A[6], 0, A[8]]])
12
13     eigenvalues, eigenvectors = np.linalg.eigh(A)
14     maximum = 0
15     index = 0
16     for i, value in enumerate(eigenvalues):
17         if value > maximum:
18             maximum = value
19             index = i
20     ei = eigenvectors.transpose()
21     point.setAttribValue("MaxVe", ei[index])
22     point.setAttribValue("MaxV", maximum*1.0)
23     i = i+1
24
25
```

Figure 17: Python Script for Calculating Separation Tensor Eigen decomposition

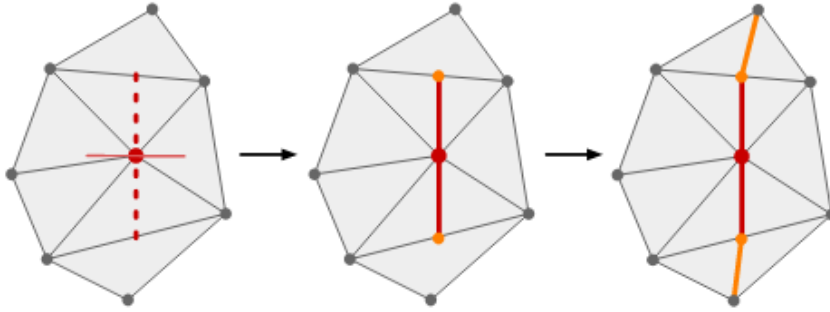
To break the script down, I iterate over each point in the geometry in the current node's geometry. Retrieves the "separation" attribute from the point (assuming it exists) and constructs a 3x3 matrix called separation from its values. I can then calculate the eigenvalues and eigenvectors of matrix A using `np.linalg.eigh()`. I find the maximum eigenvalue and use its corresponding index to set @MaxV and @MaxVe point attributes (Max Value and Max Vector). Note that the `np.linalg.eigh()` function returns the eigenvectors in the columns of the output matrix, but the indexing is by row. To get the correct eigenvectors the array needs to be transposed. At this point the eigenvalue distribution looked quite similar to the original weight field. However, of course, now we have the information for the corresponding eigenvectors for crack direction calculations and the separation tensors. Also, note the slight variation of values due to the random variation of the mesh.

### 3.6 Cracking

My methodology for cracking nodes and the visualization is based on the one presented in the original method. However, some shortcuts were taken to simplify the calculation and implementation to fit the short time period of this project. Simply put, the steps are:

1. **Determining Crack Location:** Use the highest eigenvalue of the separation tensor to decide where the crack should start. The crack then propagates in both directions orthogonal to the corresponding eigenvector. For example, for the diagram below, if the center point is the node with the highest eigenvalue, and the corresponding eigenvector is going directly to either the left or

right, the cracks should propagate directly upwards and downwards.



In the original method, new edges and points are then created to reflect this crack. However, in my methodology I snapped it toward the closest edge to simplify calculations and prevent various instabilities and artifacts mentioned.

1. **Crack Propagation to the End:** Instead of determining new crack locations for each time step, I instead chose to propagate one crack all the way to the end, as I explained before. This means that the crack keeps propagating, updating the separation tensors as in the original method, until the eigenvalue of the separation tensor is less than the material toughness. For each time a new node is chosen to be part of a crack, the separation tensor (and by extension the eigenvalues and vectors) is updated as in the original method; After cracking a node, a residual value is calculated based on the difference eigenvalue and material toughness. The separation tensor of nodes at the crack tip is modified based on this residual value and given a scaling factor of 0-1 to influence the length of cracks. The code for which is very long but can be found in my github in the Houdini file, under the node "cracking". (My Github is on the title page)

- The original method for generating cracks can be seen below. I've also included a figure and explanation as I think it is very helpful to understand how it works.

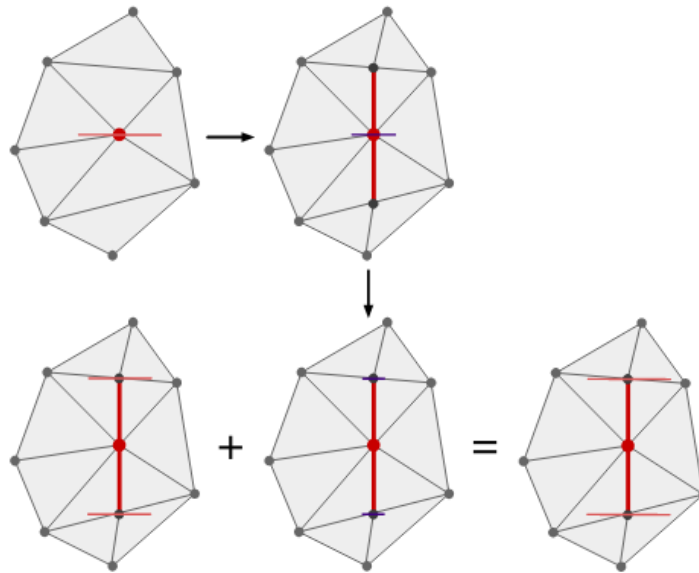


Figure 5.3: Each node has a maximum eigenvalue/eigenvector pair computed from its separation tensor (*top left*). When the maximum eigenvalue on the mesh exceeds the material toughness threshold, the algorithm inserts a crack and updates that node's separation tensor (*top right*). However, a residual amount of the maximum eigenvalue remains after the update. To account for the remaining force at the node, I multiply the residual amount by a scale factor at the crack tips (*bottom middle*) and add it to the crack tips' current separation tensor (*bottom left*) to obtain an updated tensor encouraging crack propagation in the crack's initial direction (*bottom right*). Note that I simplified this example so that the eigenvectors at the crack tips point in the same direction as the crack node; normally, the eigenvectors would vary.

*Figure 18: A Blatantly Stolen Figure. (Iben and O'Brien, 2006)*

- However, the original paper does not mention how the residual value is to be handled when a new node is created. The residual value of the original node is meant to be added to the separation tensor already present at crack tip. However, if a new node is created then it does not have a separation tensor. This means its eventual eigenvector will have a smaller value, and it will most likely not be cracked, which is exactly what happened when I implemented this remeshing technique. Whenever the crack propagation caused a new node to be created, the crack immediately ended there. Therefore, I chose to have the crack snap to the nearest edge.



2. **Relaxation:** As said in the original method: “If stress were a scalar quantity, this process would simply be mesh-based diffusion.” The basic idea behind mesh-based diffusion is to model the gradual dispersion of a scalar property over the discrete elements (nodes, cells, or vertices etc.) of a mesh. While the stress tensor is not a scalar value, the eigenvalues are. Therefore, I chose a percentage of the eigenvalue on each node (which is inevitably used to decide where the cracks are) which would be spread to the neighboring nodes. This is then done x number of times after a crack has finished propagating based on user input. It is implemented using the following code:

```
VExpression
1 float percentage = .6;
2 int neigh[] = neighbours(0, @ptnum);
3 float hold = @MaxV;
4 for(int i = 0; i < len(neigh); i++){
5     setpointattrib(0, "MaxV", neigh[i], hold*(percentage/len(neigh)), "add");
6 }
7 @MaxV = @MaxV*(1-percentage);
8
```

Figure 19: Relaxation

3. **Restart:** From there you restart the process, having updated a new node for the crack to start propagating the cracks.

To create these loops, I used loops in Houdini, as seen below. I feel that talking about all the nodes used in the loop would distract from the original method, but the original file will be on my Github.

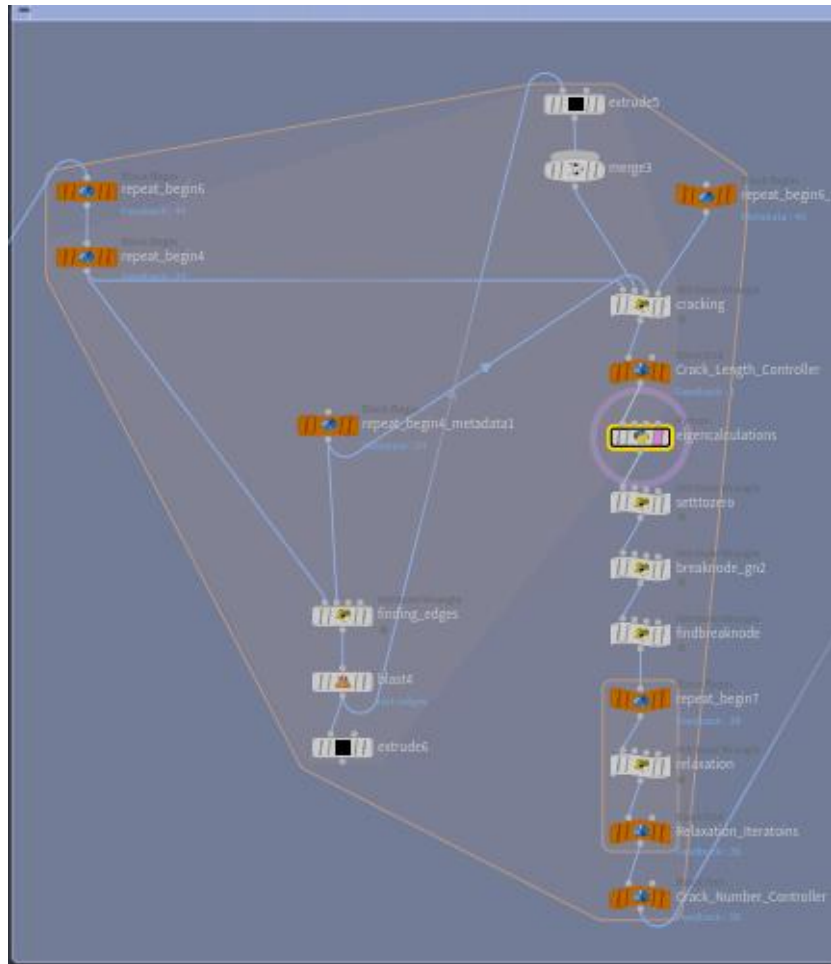


Figure 20: The main loop for calculating crack patterns.

## 4. Post Processing/Visualization

### 1.1 The Color Problem

To visualize the cracks, I set all the points that were meant to be part of a crack to black. However, since Houdini doesn't allow for setting edge colors directly within vex, this causes problems. Using the demo below, if a crack was meant to develop from the bottom-middle node to the top-middle node, then the edges between those points should also be black, as in the first photo. However, if a separate crack also develops along the edge between top-right and middle-right (second photo), then the entire primitive top right primitive between them becomes black, since all its nodes are black. When the first set of cracks were finished, this resulted in an entirely unrecognizable crack.

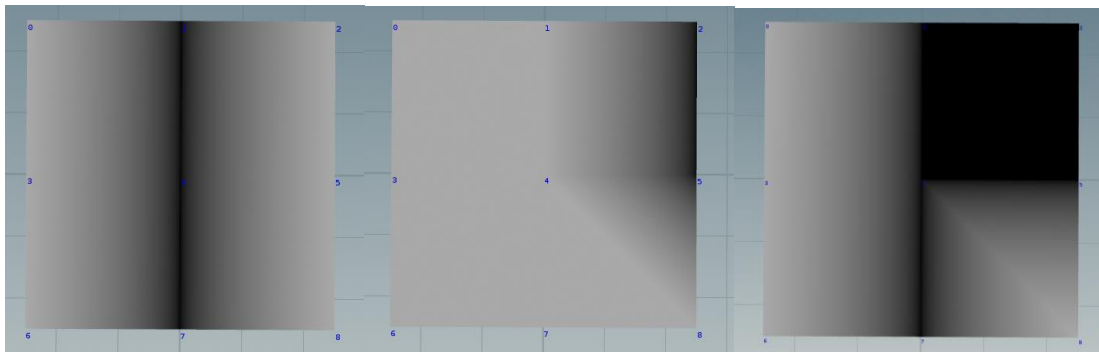


Figure 21: The Colour Problem

Instead of using a coloring method, I defined each new pair of points in the crack as a separate point group. I then created a Poly wire for each of these points, thereby visualizing where the cracks formed across each step of the process. The outcome of my simulation went from the left photo to the right photo:

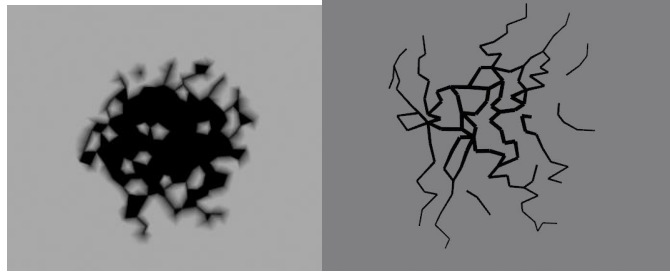


Figure 22: Visualization by setting node colors vs Polywires.

## 5. Analysis

I ran my method in Houdini and visualized it by setting the polywires to black, keeping the grey of the original plane. The run-time of the program depends on the mesh resolution and the number of cracks. For a mesh of 7,000 prims and polywire count of 238, the whole process only took about 15 seconds, varying slightly based on input attributes.

The benefits of the change in methodology allowed for a few benefits:

- By having a crack propagate all the way to the end, it allowed for more artistic control as artists can then control the maximum length of a crack. This allows for the creation of small separate cracks rather than long meandering ones such as in the example below. This can be seen in the NCCA example in section 3.1, where the cracks are all very small and short. As it turns out, this does not translate extremely well into the impact forces case, as the impact forces tend to be localized towards the impact point, causing the cracks to become very close and commonly line up.
- Because I no longer recalculate the stress tensor, I now no longer need to re-iterate through the whole process of calculating the eigenvalues of the stress tensor, positive and negative forces, separation tensor, far simplifying the calculations and speeding up the program run time.

Here are a few examples of the crack shapes generated by my crack Houdini program.

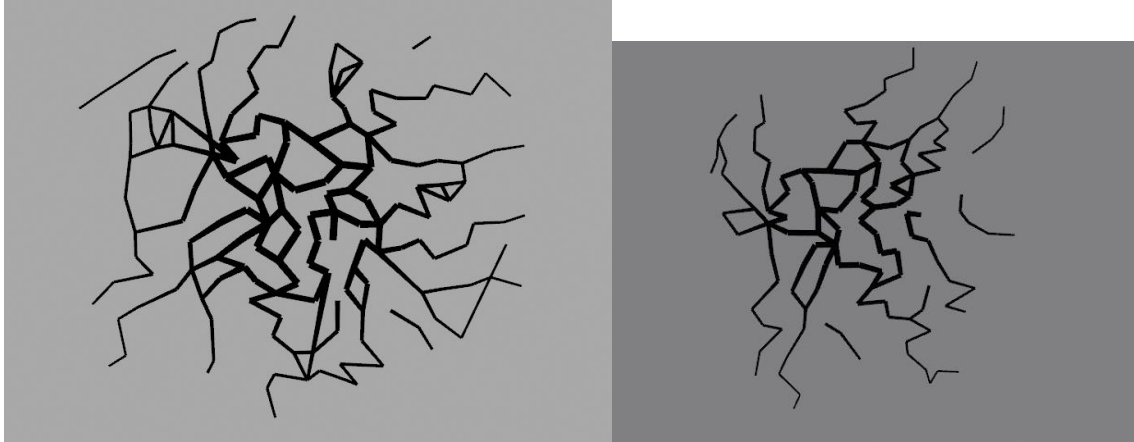


Figure 23: Final Product

Based on the success criteria, the final output had some good qualities. The cracks seemed to emanate from the center, which was indicative that the eigenvectors of the separation tensor were developing in a concentric direction as predicted. The crack shape has long meandering crack lines that split/branch, also as predicted. However, as in the pictured below, we also have long jagged lines. This is due to the mesh discretization. As my methodology does not have any remeshing, the cracks can therefore only form along the original mesh lines. If they happen to be jagged like in this example, then the only option is to have jagged lines. The fact that cracks are localized to the center is a positive in terms of realism.



Figure 24: Jagged Lines

Originally in Iben and O'Brien (2009), they introduced a post-processing step of having cracks propagate all the way until they met another crack or the edge of the geo. I applied this to my cracks manually and found a somewhat similar result. The outer lines were far more jagged than the original, again due to mesh discretization. However, the big difference found is that the original impact crack has far more clarity between concentric and radial lines. Most likely, this is due to the relaxation process presented in the paper. Using their quasi-static first-order relaxation process (essentially a diffusion process to the stress tensor), they spread out the forces, instead of having them be localized in the center. Although, the discretization could also have caused this due to the geometry not having edges that led out straight from the impact point.

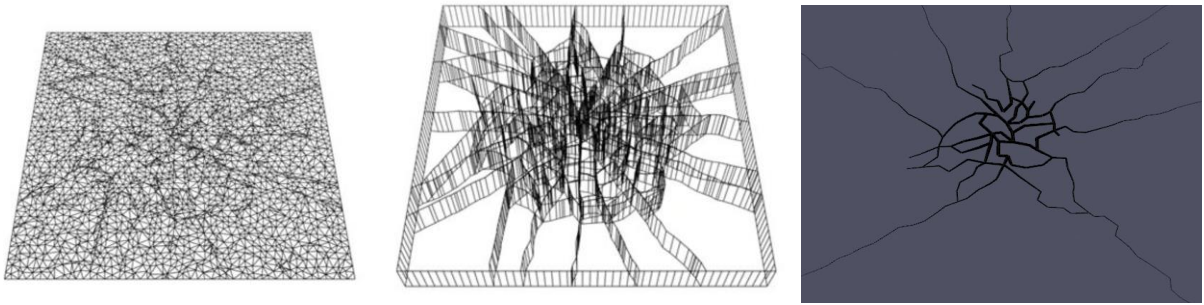


Figure 25: Comparison of final output of original method (left and middle) vs. Mine (right)

## 6. Conclusion and Future Works

In this paper I presented a novel variation to a hybrid approach for the generation of crack shapes. It creates identifiable semi-plausible cracks, with the main restraint being the mesh shape that the crack develops upon. The novel approach may therefore be called somewhat successful, although it would require more testing and development to know for sure. It passes most of the success criteria, however, looks very lackluster when compared with a real crack. It can be parametrized for easy control, and therefore would be good for development into a Houdini Digital asset. Doing so would create a reusable tool for artists to use, that would be quick and comprehensible for artists. In that respect, I think that is the best quality of this overall approach, as it has parameters that are easy to understand and clear on how they influence the outcome of the simulation. The new method also simplifies the calculations of the original method, producing quality cracks in under a minute.

### 6.1 Future Works

As it is currently, however, the implementation lacks the ability to remesh, making cracks deviate from their intended path. Although this is just a simplification of the cracking process initially proposed, it creates obvious jagged abnormalities. Although it is far more computationally simple than the original method, the relaxation only produces a noticeable effect if it is allowed to iterate many times.

Eventually, I would like to see this method developed into Houdini Digital Asset, as mentioned before it is made to be easily comprehensible in terms of input parameters and mesh. New attributes from the original method that could be given as artistic control for example would be the relaxation steps, the number of loops i.e. number of cracks that are created, the maximum length of the cracks, and the initial stress field. I think the best way to do the latter would be through painted weights or inputting impact points.





## Reference list

Alshoaibi, A.M. and Fageehi, Y.A. (2021). Adaptive Finite Element Model for Simulating Crack Growth in the Presence of Holes. *Materials*, 14(18), p.5224.

doi:<https://doi.org/10.3390/ma14185224>.

Anon, (n.d.). *STRUCTURE magazine | Crack Patterns Tell the Story of Glass Breakage*. [online] Available at: <https://www.structuremag.org/?p=17866>.

apps.dtic.mil. (n.d.). *The Application of Dynamic Relaxation to the Finite Element Method of Structural Analysis*. [online] Available at: <https://apps.dtic.mil/sti/citations/AD0676566> [Accessed 14 Aug. 2023].

caeassistant.com. (2022). *Introduction To Finite Element Method | Finite Element Analysis* ✓- *CAE Assistant*. [online] Available at: [https://caeassistant.com/blog/finite-element-method/#6\\_Limitation\\_of\\_Finite\\_Element\\_Method](https://caeassistant.com/blog/finite-element-method/#6_Limitation_of_Finite_Element_Method) [Accessed 14 Aug. 2023].

Fumoto, R. (2022). Crack Propagation using Material Point Method and J-integral. [online] p.51. Available at: <https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc22/03/CrackPropagation.pdf>.

Harvard.edu. (2023). *2x2 matrices*. [online] Available at: <https://people.math.harvard.edu/~knill/teaching/math21b2004/exhibits/2dmatrices/index.html>.

Hsieh, H.-H., Tai, W.-K., Chiang, C.-C. and Yang, M.-T. (2004). Flexible and Interactive Crack-Like Patterns Presentation on 3D Objects. *Springer eBooks*, pp.90–99. doi:[https://doi.org/10.1007/978-3-540-30182-0\\_10](https://doi.org/10.1007/978-3-540-30182-0_10).

Iben, H. and O'Brien, J.F. (2006). Generating surface crack patterns. *Symposium on Computer Animation*, pp.177–185. doi:<https://doi.org/10.5555/1218064.1218088>.

Iben, H.N. and O'Brien, J.F. (2009). Generating surface crack patterns. *Graphical Models*, 71(6), pp.198–208. doi:<https://doi.org/10.1016/j.gmod.2008.12.005>.

- Martinet, A., Galin, E., Desbenoit, B. and Samir Akkouche (2004). Procedural modeling of cracks and fractures. *HAL (Le Centre pour la Communication Scientifique Directe)*. doi:<https://doi.org/10.1109/smi.2004.1314524>.
- Matthias Müller and Gross, M. (2004). Interactive virtual materials. pp.239–246.
- Megumi Takato and Yohei Nishidate (2018). Generating Crack Patterns on Planar Geometry by L-system. doi:<https://doi.org/10.1145/3274856.3274868>.
- Muguercia, L., Bosch, C. and Patow, G. (2014). Fracture modeling in computer graphics. *Computers & Graphics*, 45, pp.86–100. doi:<https://doi.org/10.1016/j.cag.2014.08.006>.
- Müller, M., Chentanez, N. and Kim, T.-Y. (2013). Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Transactions on Graphics*, 32(4), pp.1–10. doi:<https://doi.org/10.1145/2461912.2461934>.
- Norton, A., Turk, G., Bacon, B., Gerth, J. and Sweeney, P. (1991). Animation of fracture by physical modeling. *The Visual Computer*, 7(4), pp.210–219. doi:<https://doi.org/10.1007/bf01900837>.
- O’Brien, J.J. and Hodgins, J.K. (1999). Graphical modeling and animation of brittle fracture. *arXiv (Cornell University)*. doi:<https://doi.org/10.1145/311535.311550>.
- Saty Raghavachary (2002). Fracture generation on polygonal meshes using Voronoi polygons. doi:<https://doi.org/10.1145/1242073.1242200>.
- Skjeltorp, A.T. and Meakin, P. (1988). Fracture in microsphere monolayers studied by experiment and computer simulation. *Nature*, 335(6189), pp.424–426. doi:<https://doi.org/10.1038/335424a0>.
- Sukumar, N. and Bolander, J. (n.d.). *Voronoi-based Interpolants for Fracture Modelling*. [online] Available at: <http://dilbert.engr.ucdavis.edu/~suku/nem/papers/voronoifracture.pdf> [Accessed 14 Aug. 2023].

Swenson, D.V. and Ingraffea, A.R. (1988). Modeling mixed-mode dynamic crack propagation using finite elements: Theory and applications. *Computational Mechanics*, 3(6), pp.381–397. doi:<https://doi.org/10.1007/bf00301139>.

Terzopoulos, D. and Fleischer, K. (1988). Modeling inelastic deformation. *ACM SIGGRAPH Computer Graphics*, 22(4), pp.269–278. doi:<https://doi.org/10.1145/378456.378522>.

Tiwari, N., Harshey, A., Das, T., Abhyankar, S., Yadav, V.K., Nigam, K., Anand, V.R. and Srivastava, A. (2019). Evidential significance of multiple fracture patterns on the glass in forensic ballistics. *Egyptian Journal of Forensic Sciences*, 9(1). doi:<https://doi.org/10.1186/s41935-019-0128-4>.

Valette, G., Stéphanie Prévost and Lucas, L. (2006). A generalized cracks simulation on 3D-meshes. *Eurographics*, pp.7–14. doi:<https://doi.org/10.5555/2381370.2381372>.

Weisstein, E.W. (n.d.). *Barycentric Coordinates*. [online] mathworld.wolfram.com. Available at: <https://mathworld.wolfram.com/BarycentricCoordinates.html>.