Universität Konstanz
FB Informatik und Informationswissenschaft
Master-Studiengang Information Engineering

Masterarbeit

Ein Stressmodell zur Simulation der Entwicklung von Rinde

A Stress Model for Simulating Tree Bark Development

zur Erlangung des akademischen Grades eines
Master of Sciene (M.Sc.)

Studienfach:	Information Engineering
Schwerpunkt:	Computergrafik
Themengebiet:	Echtzeit Computergrafik

von

Marc Spicker
(691558)

Erstgutachter:	Prof. Dr. Oliver Deussen
Zweitgutachter:	Jun.-Prof. Dr. Michael Grossniklaus
Betreuer:	Julian Kratt, Dr. Sören Pirk
Einreichung:	April 14, 2014

Declaration of Authorship

The author of this work hereby declares that

- the present work is the result of his own work, without help from others and without using anything other than the named sources and aids;

- the texts, illustrations and/or ideas taken directly or indirectly from other sources (including electronic resources) have without exception been acknowledged and have been referenced in accordance with academic standards.

This work has been created in context of a joint research project between Purdue University (USA), University of Illinois (USA) and the University of Konstanz (Germany). The work described in Section 5 has been done by Purdue (Bedrich Benes, Alejandro Guayaquil Sosa, Marek Fiser) and Illinois (John C. Hart), my contribution is the coupling between the evolving meshes and the surface cracking method described in Section 3.

The Author wants to gratefully acknowledge supervision and guidance he has received from Sören Pirk and Julian Kratt.

Konstanz, April 14, 2014 Marc Spicker

Abstract

In computer graphics a large amount of work has been invested into the realistic display of vegetation. Application areas range from fields such as architecture over computer games to computer animation for films. Since trees have the most prominent appearance in vegetation, a large focus lies on the faithful representation of synthetic trees. Works done on the rendering of trees often target either the structural representation, the foliage, or the tree trunk. The presented work focuses on the latter. A simulation method is developed that recreates the surface appearance of tree bark defined by imperfections therein: cracks. It incorporates an already established physically-based stress model which evolves a stress field over time. The result of this model is rendered with current graphics hardware to create compelling images resembling natural trees at interactive frame rates. This technique is also applied to evolving objects, allowing to simulate a natural growth process.

Contents

1 Introduction **1**

2 Related Work **4**

3 Bark Generation **7**
3.1 Surface Cracking Model . . . 7
3.1.1 Stress Field . . . 8
3.1.2 Force Calculation . . . 9
3.1.3 Cracking . . . 11
3.1.4 Relaxation . . . 13
3.1.5 User Control . . . 15
3.1.6 Post Processing . . . 16
3.2 Stress Alleviation . . . 17
3.3 User-based Stress Initialization . . . 18

4 Bark Rendering **21**
4.1 Normal Mapping . . . 21
4.2 Displacement Mapping . . . 24

5 Evolving Meshes **30**
5.1 Introduction . . . 30
5.2 Related Work . . . 30
5.3 Deformable Simpicial Complex Method . . . 32
5.3.1 Allometric Growth . . . 34
5.3.2 Structural Growth . . . 34
5.3.3 Reversed Growth . . . 35
5.3.4 User-defined Growth Sketching . . . 35
5.3.5 Growth Around Obstacles . . . 35
5.4 Evolving Meshes with Surface Cracking . . . 36
5.4.1 Texture Coordinate Propagation . . . 37
5.4.2 Transfer of Cracks . . . 38
5.4.3 Stress Field Transfer . . . 38
5.4.4 Stress by Growth . . . 39
5.4.5 Technical Overview . . . 39

6 Results **40**

7 Conclusion **50**

8 Future Work **51**

1 Introduction

In nature we encounter materials with complex surface structures. Most materials change due to ageing, growth or outside influences. They break, change their color or become cracked. Those processes define the appearance of these kinds of materials and therefore the human eye is sensitive to them. Some examples for such materials can be seen in Figure 1: dried mud, broken concrete and rusty metal with peeling paint. Dried mud forms cracks because the mud on the surface dries faster than the underlying due to the direct sunlight. This process creates stress which is relieved by cracks. Concrete cracks mainly because of temperature changes which causes it to expand and shrink. When this happens too fast, the material can fail. Painted metal starts to rust when water penetrates the paint in some areas. The rust will grow under the paint into different directions which causes more paint to peel.

Figure 1: Different materials which define their appearance by the surface structure: Dried mud (image Frank Vincentz), broken concrete (image Dmitriy Chugai) and rusty metal (image `fantasystock.deviantart.com`).

In computer graphics the objects are usually defined with simple, flawless surfaces. We can easily distinguish between realistic surfaces and those appearing too perfect and therefore synthetic. For applications like animation, feature films and computer games visually compelling images are needed. Much work has been dedicated to this topic, some of which is described in Section 2. Instead of physical faithfully simulating the processes behind the creation of the different surface structures, they aim at creating plausible images. While an artist can create compelling results, the amount of work is exhausting and the result is very specific. It might have to be redone if something slightly different is wanted. Trees are an excellent example of this: different species vary greatly in their surface appearance, as seen in Figure 2. An automatic tool with some amount of artistic control is wanted to reduce the artistic work while maintaining the possibility to control the result. The presented work focuses on creating visually appealing surface patterns for different types of tree bark.

Figure 2: Tree bark is mostly defined from its surface structure. It greatly varies between different species. The example shown are from a palm, a young birch and a pine.

Appearance of bark can be divided into two main parts: high-frequency information like small patches of moss or the non-uniform color of the bark and low-frequency structures. These low-frequency structures are mainly due to cracks in the bark that appear during the growth of the tree. When a tree grows, a thin layer under the bark (the *phellogen*) creates wood towards the inside and bark towards the outside. The circumference of the tree increases resulting in stress within the bark which can cause fractures. Since the high-frequency information can be applied with established techniques like texture mapping, the focus lies on generating low-frequency information from the surface cracks without a large amount of user input. With the ability of simulating crack patterns it is also possible to model the ageing of a tree by changing the bark, since it is the main (exterior) hint at the age of a tree. Methods which aim at generating such low-frequency information can be grouped into two main categories: non-physical and physical approaches. Both are explained in greater detail in Section 2. To create surface crack patterns for tree bark, a more general model for surface crack patterns based on a physical stress-based approach is taken and modified for this purpose, as described in Section 3. The presented method creates compelling results for different types of tree bark while still allowing the user a large amount of control over the general appearance of the cracks. Some of these results can be seen in Section 6. The underlying physical model does not allow these results to be generated in real-time, however the rendering can be done at interactive frame rates. In Section 5 this method is applied to non-static evolving meshes to simulate wood-like objects. A cambial growth model is introduced based on mesh surface propagation which allows to capture a variety of natural growth processes. It also includes obstacle collision and botanically motivated response. The growth of the object introduces or increases existing stress on the surface, depending on the strength of the growth. This stress is then used for the surface cracking algorithm to model cracks during to the growth process. Advantages and disadvantages of the presented system are discussed in Section 7 and some ideas for future work are given in Section 8. There are three main contributions presented

in this work: An existing model for simulating surface cracks is adapted for the purpose of generating the cracks appearing in tree bark. The second contribution is the developed rendering technique which allows for a faithful display of surfaces with the appearance of tree bark. It builds upon the previous mentioned technique for crack simulation. The coupling of these methods with a mesh evolution technique is the final contribution which enables to grow input objects in a natural manner.

2 Related Work

Tree bark modelling has developed from simple approximative techniques to more biological faithful simulations. Some of the early works in this field parametrized a simple branching tree model to support a texture mapping which incorporates the directions of the branching structures [Bloomenthal, 1985]. Another method for surface parametrization uses a particle flow from the trunk to the branches of a tree [Hart and Baker, 1996]. More recently, a lot of work has been done in the simulation of the surface structure of trees: cracks. The field of crack creation and material failure has been intensively studied in fields such as physics, material science and geology. The work described in this section focuses on those presented in computer science, especially those in the field of computer graphics. Furthermore, primarily works with the purpose of modelling tree bark or capable of simulating tree bark are discussed. In computer graphics these approaches can be divided into physical and non-physical approaches. An example for a non-physical approach to simulate tree bark is based on image processing [Wang et al., 2003]. Given a single input image, texton channel analysis is applied to find structures typical for different types of bark. The input image is then segmented into different channels for different features. This is done by applying a bank of Gaussian derivative filters for each pixel resulting in a high-dimensional data vector which is clustered by the K-means algorithm [Gersho and Gray, 1991]. Pixels grouped together belong to the same channel. By merging these channels, the user can create a custom surface appearance that is desired. Figure 3 shows the segmented texton channels on top. The bottom shows the input image on the left, the merged channels for fractures in the middle and for lenticel on the right. In a separate step a height map is created interactively from this merged image by specifying the depth of certain regions of the bark. A problem of this approach is that it requires a lot of user interaction and it only offers a limited flexibility to the creation of custom cracks due to the input image. On the side of the physically-based simulations, one approach divides a surface area into strips parallel to the

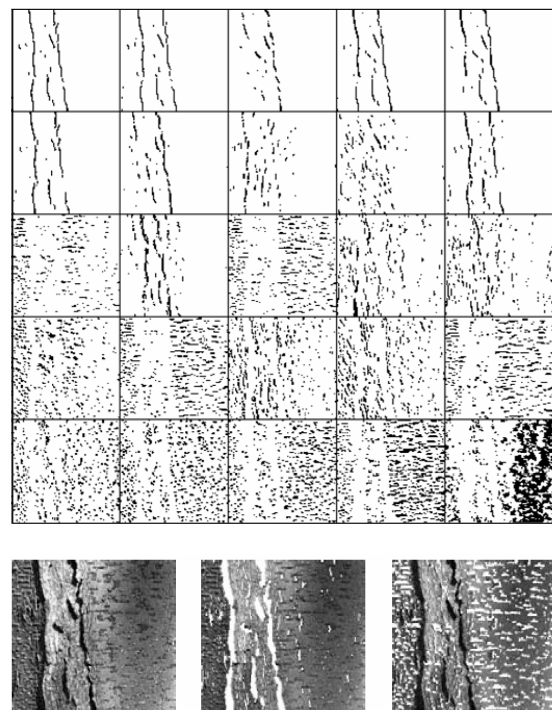


Figure 3: Top: segmented texton channels. Bottom: input image left, merged channels for fracture middle, for lenticels right.

growing direction depicted in Figure 4 [Lefebvre and Neyret, 2002]. The strips are handled like springs and Hooke's law is used to model their behaviour. Growth is simulated by extending these springs. A linear equation system with the stiffness values of the springs is solved to distribute the lengthening over the springs in a strip. When the lengthening of the springs exceeds a threshold, a crack occurs and a new spring with the length equal to the fracture width is added at this position with a different stiffness corresponding to the fracture material. These cracks can propagate into adjacent strips to simulate the propagation of existing cracks. Since the circumference of a tree changes with regard to the height, the width of the strips is initialized accordingly. After the simulation, the strips are used to generate geometry according to the cracks. The model can also simulate the attachment of the bark to the underlying substrate by inserting additional springs from a fixed point on the substrate to the springs in the strips. It is also possible to provide textures for the bark and fracture to synthesize a new texture incorporating the cracks.

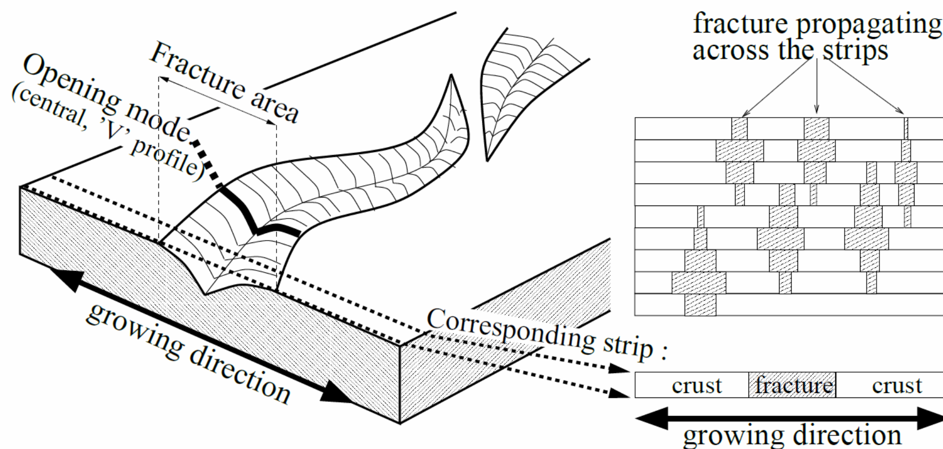


Figure 4: A surface area is divided into parallel strips. Elements in the strips are modelled as springs [Lefebvre and Neyret, 2002].

In another approach, fractures are simulated with a bi-layered model: a volumetric material layer with prism elements, depicted in Figure 5, and a background layer to which the lower vertices of the prisms are attached [Federl and Prusinkiewicz, 2002]. Growth is modelled by moving the attachment points on the background layer which introduces stress in the prisms. When the stress exceeds a material threshold, a prism element is removed resulting in an open crack. The method also allows dynamic subdivision of the elements so that the fracture size created by the removal of an element can be better controlled. A different volumetric method uses tetrahedral elements instead of prisms [O'Brien and Hodgins, 1999]. Stress tensors are defined per tetrahedron in a FEM approach which apply forces to the vertices of the mesh. If the stress at the vertices exceeds a material threshold due to deformation, a crack occurs. A fracture plane is defined based on the eigenvectors of

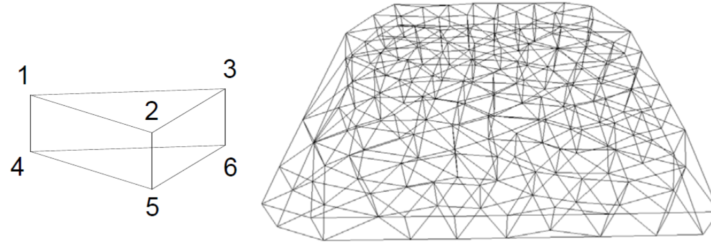


Figure 5: Prism elements in the volumetric material layer.

the stress tensor at the point of material failure. This fracture plane splits intersecting tetrahedrons. Since the polyhedra resulting from the intersection with the splitting plane does not necessarily produce new tetrahedrons, these polyhedra have to be decomposed into tetrahedra in a second step. Three fracture modes can be simulated with this method as seen in Figure 6: opening, in-plane shear and out-of-plane shear. The system is also

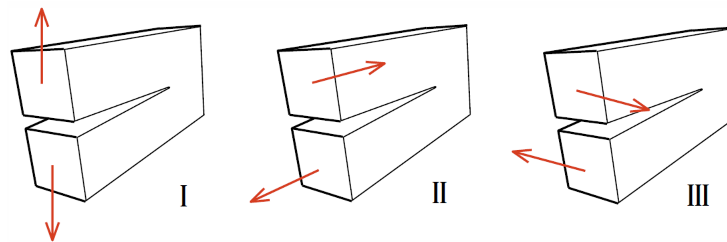


Figure 6: Three different modes of fracture: Opening (I), in-plane shear (II) and out-of-plane shear (III).

capable of handling collisions. A penalty is applied to the forces when two elements intersect or other constraints, like the ground, are violated. The penalty is proportional to the overlapping area. This paper builds the foundation for a faster non-volumetric approach [Iben and O'Brien, 2006] on which the method presented in this work builds upon. A surface discretization is used instead of a volumetric mesh and the stress field is initialized directly through heuristics. The main reason for choosing this method is that none of the other approaches offers enough flexibility in the appearance and control over the cracks to make it suitable for the rendering and simulation of cracks in tree bark. Cracks resulting from the other approaches resemble those in solid materials like glass which is not realistic for those biologically occurring in trees. The large amount of parameters to control the visual appearance of the cracks can be used to create different cracks for different tree species which can tremendously differ in their surface crack appearance. The increased speed-up by using a surface mesh is more than welcome when applying this method to evolving meshes. In Section 3, the existing work is described with the changes applied to make it suitable for the simulation and rendering of tree bark.

3 Bark Generation

The tool for simulating tree bark development builds upon a physically-based surface cracking model [Iben and O'Brien, 2006][Iben, 2007]. This section gives an in-depth technical description of the method and is structured in the following way: Section 3.1 describes the underlying method and in Section 3.2 a newly added feature to the cracking model is introduced which allows for further control over the evolution of the stress field. In Section 3.3 two new methods for initializing the stress field are proposed.

3.1 Surface Cracking Model

In contrast to many crack simulation algorithms, the proposed method works on a 2D surface triangle mesh discretization instead of a volumetric representation. An example for such a surface triangle mesh can be seen in Figure 7. A stress field is defined over the triangle mesh which is evolved over time. Stress is distributed from regions of high stress to regions of low stress and if the stress exceeds a threshold, cracks occur in the mesh alleviating stress in the surrounding area. The triangle mesh is altered by adding the crack edges into the mesh. The user has full control over the appearance of the cracks by a set of parameters. This makes the proposed method well suited for an artistic tool. An overview of the complete algorithm can be seen in Figure 8. It starts with an initialization step in which the user can define the stress field over the triangle mesh. A simulation loop consisting of three steps follows: The forces resulting from the stress field are calculated in the first step, possible material failures and cracks are determined in step two, and in the last step stress is distributed from regions of high to regions of lower stress. After a predefined number of iterations or after the user is satisfied with the result, the created cracks have to be rendered in a post-processing step. This technique can also be used for time-lapse animation, since the minimum amount a crack edge grows is the length of the next edge in the mesh and therefore the transition will never be completely smooth. In the following, each step of the algorithm is described in detail.

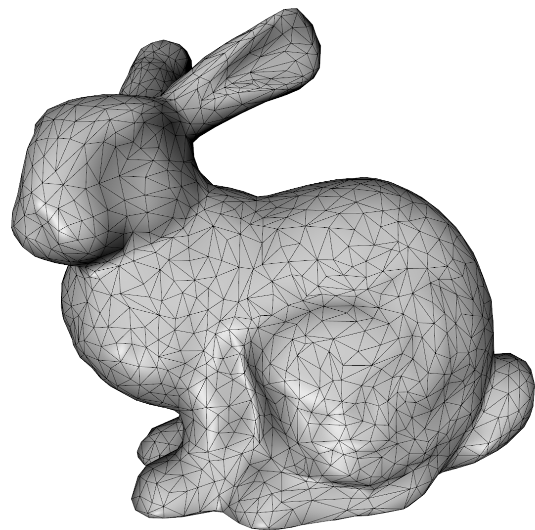


Figure 7: An example triangle surface mesh.

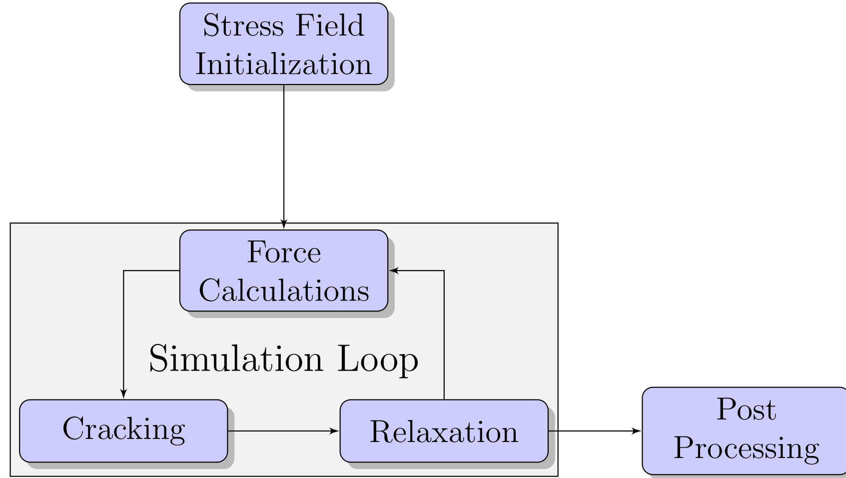


Figure 8: Overview of the presented algorithm.

3.1.1 Stress Field

Physical stress can be decomposed into two main parts illustrated in Figure 9: tensile and compressive. Tensile stress pulls apart from a point into opposite directions and a possible result of this process can be an open facture at this point. Compressive stress pushes towards a point from different directions. Stress has no direction and can be stored in matrices, the stress tensors σ. We will focus on two dimensional stress and therefore the stress tensor is a 2×2 matrix. The eigenvector has the principal direction of the red line and the eigenvalue α determines whether the stress is tensile ($\alpha > 0$) or compressive ($\alpha < 0$). Assuming symmetry, a 2×2 matrix can be decomposed into two real eigenvector and eigenvalue pairs and therefore the total stress can be composed of any combination of tensile and/or compressive parts. This holds because every proposed way of initializing the stress field (and therefore the stress tensor) results in symmetric matrices. The stress field is defined as a stress tensor per triangle of the input mesh. Positions of the triangle vertices v_i are defined in global 3D world coordinates and to reduce the dimensions, the stress tensor is defined in the local 2D coordinate frame *(u,v)* defined by a triangle edge and the cross product between the edge and the normal of the triangle depicted in Figure 10. These local coordinate frames can be stored in the triangle so that they only have to be calculated once. The normal of the triangle is given by the input mesh. The local coordinates of points in this coordinate system can be calculated as the length of the vectors u and v.

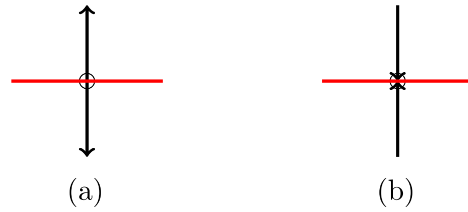


Figure 9: Decomposition of stress into tensile **(a)** and compressive **(b)** stress.

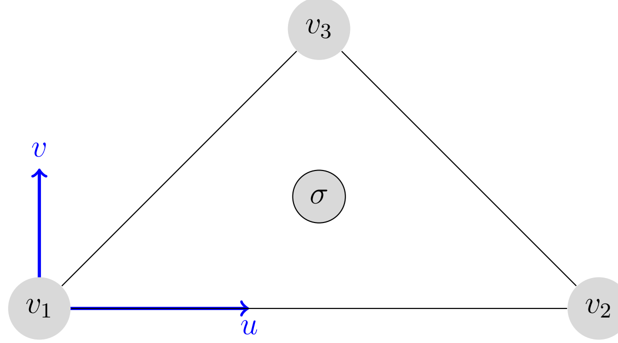


Figure 10: The local 2D coordinate frame (u,v) in which the stress tensor σ is defined.

3.1.2 Force Calculation

The stress tensor σ in each triangle applies forces to its vertices. These forces would result in a movement, but instead the positions of the vertices are kept static and the displacement is simply stored. This avoids numerical problems in the integration, since imprecisions would add up over the time of the simulation. The force $f_{[i]}$ applied on the node v_i from the stress tensor can be determined by

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

where A is the area of the triangle, $p_{[i]}$ the 3D world space coordinate of vertex v_i and β a 3×3 matrix which will be derived later on. The forces for a single triangle can be seen in Figure 11.

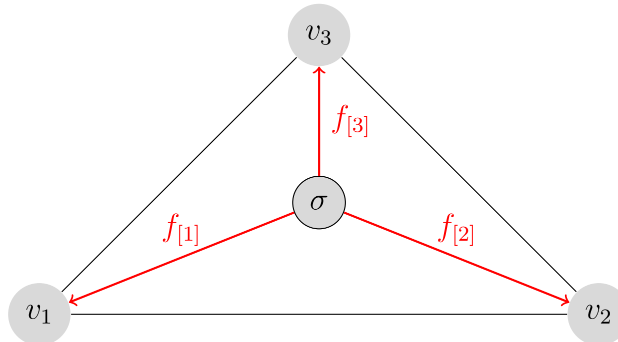


Figure 11: The stress tensor σ in each triangle applies forces to each of its vertices.

β is called the *barycentric basis matrix* [O'Brien and Hodgins, 1999]. It defines a mapping from local 2D coordinates (u, v) to barycentric coordinates. It can be constructed from the local coordinates m_{iu}, m_{iv} of each vertex v_i:

$$\beta = \begin{pmatrix} m_{1u} & m_{2u} & m_{3u} \\ m_{1v} & m_{2v} & m_{3v} \\ 1 & 1 & 1 \end{pmatrix}^{-1}$$

This describes the shape of the triangle which is similar for similarly shaped triangles and dissimilar otherwise. Until now, we have calculated the forces for each triangle of the input mesh locally: a triangle applies one force to each of its vertices. On a global mesh scale however, triangles share common vertices. A vertex can have multiple adjacent faces, assuming a valid triangulation. Each face contributes a force acting on the vertex. As previously stated, stress can be decomposed into a tensile and compressive part. The stress tensor σ can also be decomposed into tensile σ^+ and compressive σ^- parts:

$$\sigma^+ = \sum_{i=1}^{2} max(0, v^i(\sigma))m(\hat{n}^i(\sigma)) \tag{2}$$

$$\sigma^- = \sum_{i=1}^{2} min(0, v^i(\sigma))m(\hat{n}^i(\sigma)) \tag{3}$$

where v^i is the i-th eigenvalue, $\hat{n}^i$ is the i-th eigenvector and m is the outer product of a vector which is defined as $v \cdot v^T$. The decomposed stress tensors σ^+, σ^- can be inserted into Formula 1 to calculate the tensile (f^+) and compressive (f^-) forces. From a vertex point of view, each adjacent face now contributes two forces, which can be seen in Figure 12.

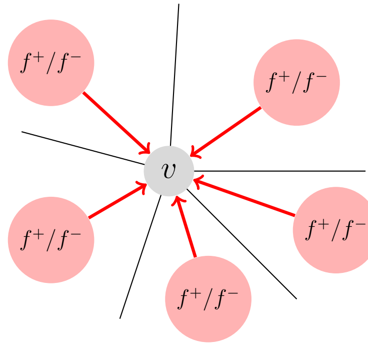


Figure 12: Each adjacent face applies forces to a vertex. These forces can be decomposed into tensile (f^+) and compressive (f^-) parts.

The tensile and compressive forces from the adjacent faces can be used to calculate the total stress acting on a vertex, the *separation tensor* ς:

$$\varsigma = \frac{1}{2}(-m(f^+) + \sum_{f \in \{f^+\}} m(f) + m(f^-) + \sum_{f \in \{f^-\}} m(f)) \tag{4}$$

where m is again the outer product of a vector. In the force calculation step we have transferred the stress, which was initialized in each triangle, onto the vertices of the triangle mesh. In the following step we use this stress per vertex to determine possible points of material failure and therefore cracks.

3.1.3 Cracking

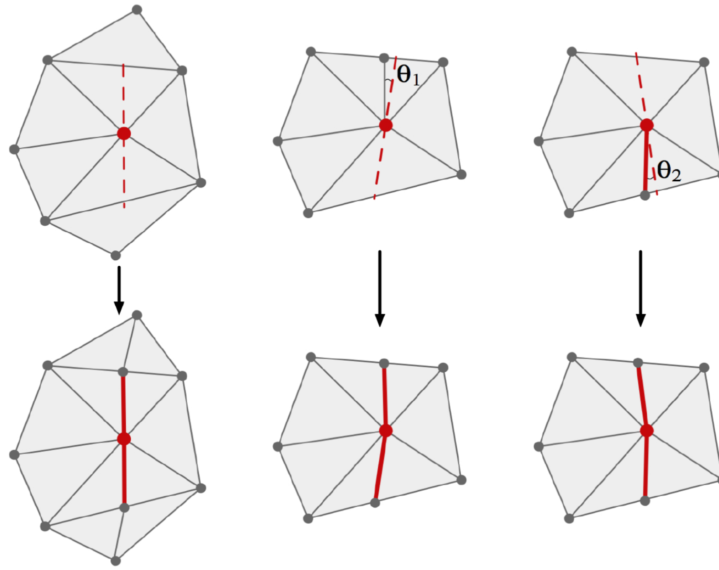


Figure 13: Left: crack edges (red lines) are inserted at the cracking plane (red dashed line) perpendicular to the eigenvector of the crack vertex (red node). Additional edges have to be inserted to ensure a valid triangulation. Center and Right: to avoid sliver triangles, the crack edge ‘snaps’ onto an existing edge if the angle in between is under a threshold.

The separation tensor calculated in the force calculation step determines the stress acting on each node of the input triangle mesh. In this step, the stress is used to decide whether the material fails at the position of the nodes and if so, introduce new crack edges into the mesh. The eigenvalues of each separation tensors are calculated and inserted into a priority

queue. If the largest positive (tensile) eigenvalue at the top of the priority queue is larger than a user defined material threshold τ, a crack occurs at this vertex. The direction of the crack is perpendicular to the corresponding eigenvector. To introduce more variety, the user can rotate the created crack edge by a random amount. Crack edges are inserted into the mesh in this direction up to the point where they reach the boundaries of the triangle they started in. This can be calculated by a single ray-line intersection. To ensure a valid triangulation, additional edges may have to be inserted into the mesh. These edges are not marked as crack edges. This is illustrated for a small triangle mesh in Figure 13: On the left the vertex with a eigenvector of the separation tensor exceeding the material threshold is marked as red dot and the direction perpendicular to the corresponding eigenvector is marked as red dashed line. Two red crack edges are inserted into the mesh in this direction and two addition edges have to be inserted at the top and bottom triangle to ensure a valid triangulation. New crack edges can be close to existing edges in the mesh, resulting in sliver triangles which in the later step of the algorithm pose a problem to the numeric stability thereof. To avoid these, the angle between the crack direction and the existing edges are determined. If under a user given threshold, the crack edge ‘snaps’ onto the existing edge, using it as crack edge which is depicted in the middle of Figure 13. The same holds true for the other side of the crack, as seen on the right. The crack appearance can be controlled by adding additional stress to the separation tensor at the crack tip: the *residual value* v^*. It is defined as the surplus of the eigenvalue and the material threshold:

$$v^* = v^+ - \tau \tag{5}$$

The separation tensor is updated by adding the outer product of the tensile parts eigenvector times a user defined value $\alpha \in [0, 1]$, which controls the weighting of the residual value.

$$\varsigma' = \varsigma + \alpha m(\hat{n}^+) v^* \tag{6}$$

Large values for α result in larger cracks which move further in the same direction. This is due to the high amount of stress added to the crack tips, which makes them more likely to be a crack vertex in the next iteration of the algorithm. Smaller values for α result in smaller, more jagged edges. Results for different residual value weightings can be seen in Figure 14, from a low weighting on the top left to high residual values on the bottom right.

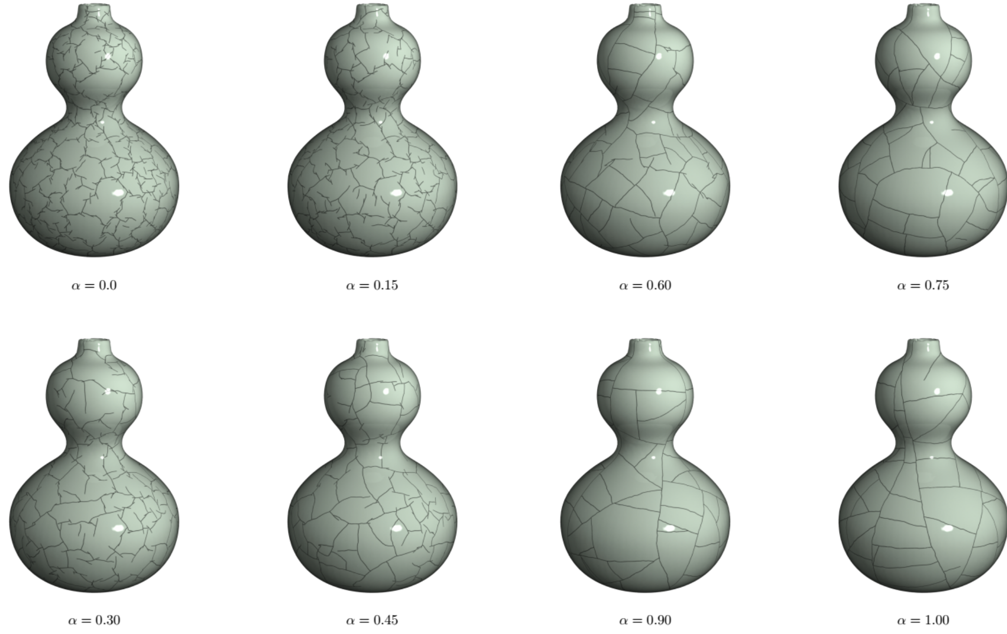


Figure 14: Cracking results for different weightings of the residual value α. Small values result in small, jagged edges which can be seen on the left, whereas large values create longer cracks which move further in the same direction, seen on the right.

3.1.4 Relaxation

In the relaxation step of the algorithm, stress is distributed from areas with high stress to areas with lower stress. This mimics natural diffusion processes like temperature and air pressure and allows for more evenly distributed cracks over the surface area. Stress is treated as an independent variable and if stress were a scalar quantity, the process would be mesh-based diffusion. However, since stress is a tensor and is defined locally in each triangle, the diffusion is more complex. To calculate the change in stress $\Delta\sigma$, two quantities have to be determined: on the one hand side the forces acting on a node from the surrounding elements, and on the other hand the deformation that the triangle has undergone from these acting forces. For the numerical stability during the integration, forces acting on the nodes do not actually result in a movement of the nodes. These forces can be summed up as the *virtual displacement*:

$$\Delta p_{[n]} = \Delta t \; F_{[n]} \; , \tag{7}$$

where $F_{[n]}$ is the sum of all forces $f_{[n]}$ of the surrounding elements acting on a node and Δt is a time step that controls the rate of relaxation. The deformation of the triangle can

be described by *Green's Strain tensor* ε [Fung, 1965], a well known quantity in material science. Since we are interested in the development of the strain over time, we can use the derivative with regard to the node position to finish the complete formula, assuming a linear relationship between stress and strain:

$$\Delta\sigma = \frac{\partial \varepsilon}{\partial p_{[n]}} \Delta p_{[n]} \ . \tag{8}$$

The first derived part can under the same assumption be substituted to

$$\frac{\partial \varepsilon_{ij}}{\partial p_{[n]_r}} = \frac{1}{2} \left(\sum_{m=1}^{3} p_{[m]_r} \beta_{mi} \beta_{mj} + \sum_{m=1}^{3} p_{[m]_r} \beta_{mi} \beta_{mj} \right) \ , \tag{9}$$

which only contains already known quantities from the stress initialization. An example for this process can be seen in Figure 15. On the left side a triangle mesh is shown in which the eigenvalue of the stress tensor is depicted as length of a bar in the center of the triangle, the direction equals the corresponding eigenvector. A high amount of stress exists at the center of the mesh, whereas on the right and left side it equals zero. The stress after applying the relaxation algorithm can be seen on the right. The areas without stress now have a small amount of stress and the areas with high stress now have a lower stress.

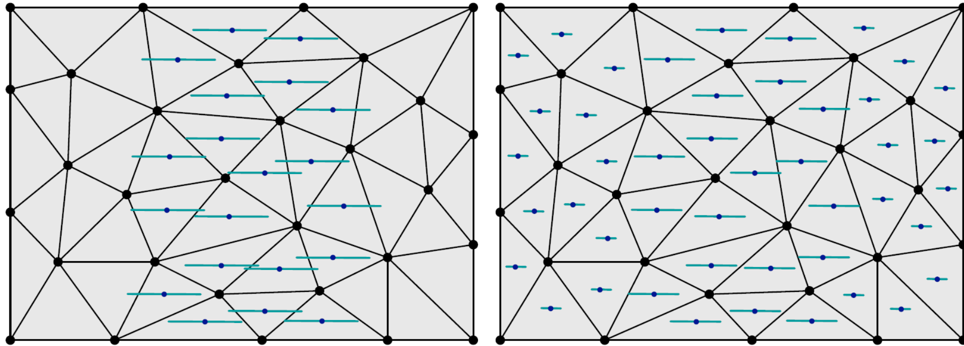


Figure 15: An example triangle mesh before and after relaxation. Eigenvalues of the stress tensors are depicted as length of bars in the center of the triangles, the direction is determined by the corresponding eigenvector.

The numerical stability issues with sliver triangles mentioned in the cracking section can be tracked back to the relaxation process. Large singular values in the barycentric basis matrix created by degenerate triangles produce large values in the stress tensor update $\Delta\sigma$, which are added up onto the stress tensor over the iterations of the algorithm. To

dampen this problem, the maximal singular value of all barycentric basis matrices of the initial mesh is initially calculated via singular value decomposition. This value times a user defined factor κ is used to limit the maximum possible singular value that can occur in newly created triangles. Of course it highly depends on the triangles of the input mesh. The quality of the input mesh has a large influence on the result of the algorithm: regular meshes result in unnatural regular crack patterns and pre-existing sliver triangles can pose problems to the numerical stability that cannot be resolved with the proposed dampening technique. An example for the different results on a regular and irregular mesh can be seen in Figure 16.

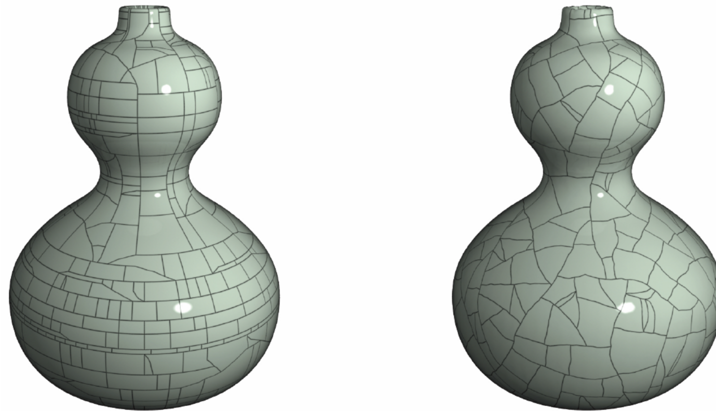


Figure 16: Left: algorithm run on a regular mesh creates a unnatural regular appearance. Right: algorithm with same parameters on irregular mesh.

Therefore the initial meshes are preprocessed if they are too regular or have degenerated triangles. The regularity is removed by applying a re-tiling method [Turk, 1992]. We found the following preprocessing steps to produce good results: as a first step, points are sampled with a Poisson disk sampling on the surface of the triangle mesh. In a second step, a surface is reconstructed by the ball pivoting algorithm [Bernardini et al., 1999]. This results in non-sliver triangle with similar area, which are irregular due to the random sampling strategy.

3.1.5 User Control

Multiple steps in the algorithm have introduced parameters with which the user can directly influence the outcome of the algorithm. Figure 17 shows which parameter control/influence which step of the algorithm. In the stress field initialization step, the user can freely define the stress tensor σ for each triangle. In the original paper several predefined patterns like horizontal and vertical stress and uniform tension are proposed, but arbitrary stress patterns can be used. The appearance of the cracks can be controlled by the weighting of

the residual value α, which determines whether cracks are long, or short and jagged. The crack edges can also be rotated by a random amount to introduce more variety in the crack patterns. In the relaxation step the rate of relaxation can be controlled by the parameter Δt and the dampening factor κ aims to reduce the numerical instability introduced by sliver triangles at the cost of accuracy in the stress change calculation $\Delta\sigma$.

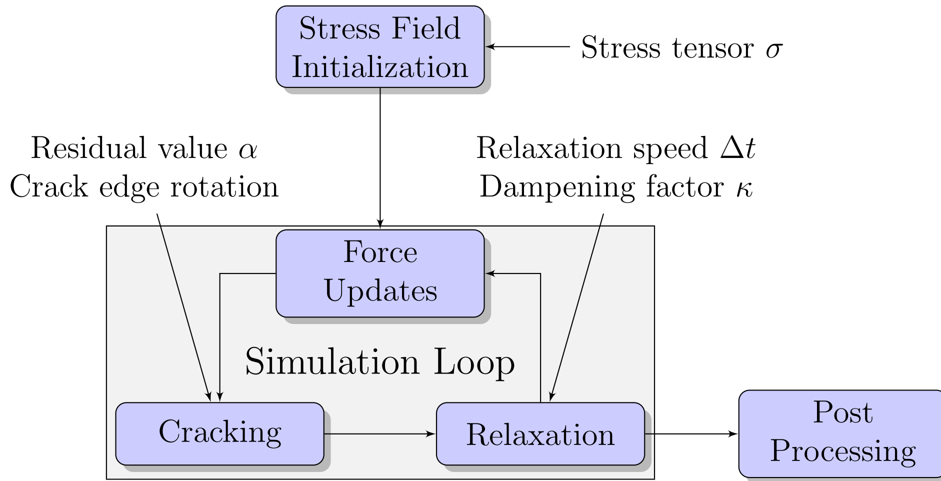


Figure 17: The user control over the different steps of the presented method and the corresponding parameters.

3.1.6 Post Processing

Up to now, the cracks are simply edges in the mesh. To make them visually apparent, they have to be post-processed in a separate step. In the original paper, two methods to render these cracks are suggested. One way is to render them directly in a different colour. To achieve ceramic-like cracks, a gradient can be applied to the line, seen in Figure 18. Since the line rendering and the shaded faces have the same depth value, a polygon offset has to be defined to avoid z-fighting artefacts. If however a visual gap between the crack sides is wanted, the mesh has to be changed at the cracks. One possible way to create a gap is to insert the crack edge twice and push them apart. The change of each node position during the relaxation process is summed up to determine the width of the gap. The result for an example crack can be seen in Figure 19 where the crack edges are marked red and the arrows depict the summed up virtual displacement during the relaxation process. Both techniques do not add any depth to the cracks. A mesh created with the second technique can be extruded to create a very simple perception of depth. In Chapter 4, more advanced rendering techniques based on displacement mapping are proposed to create more visually appealing results which also deals with the problem of depth.

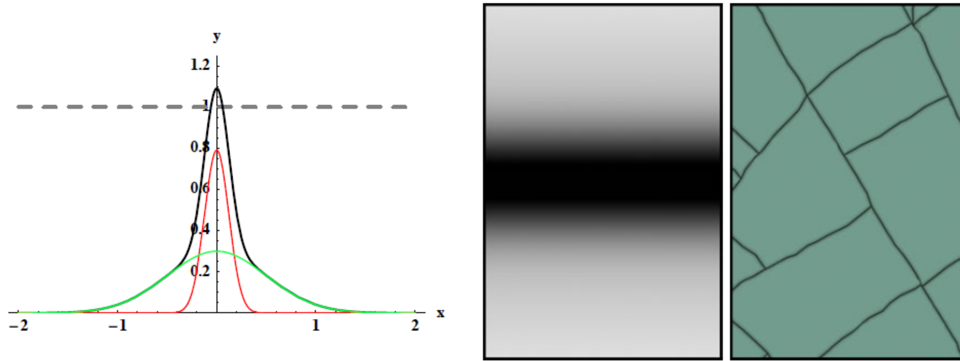


Figure 18: When rendering crack edges directly, a gradient can be applied to the line to create a ceramic-like crack resemblance.

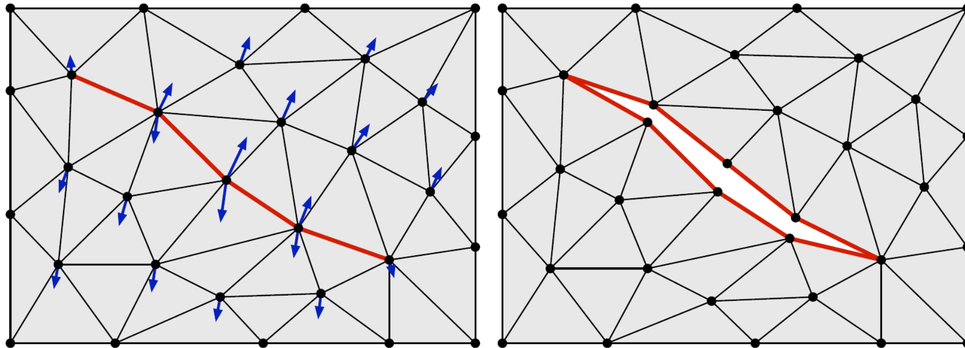


Figure 19: A visual gap is created by changing the mesh at the crack location. The width of the gap is determined by the summed up movement of each node during the relaxation process.

3.2 Stress Alleviation

When the stress in a material exceeds the material-specific threshold and fails, cracks occur which alleviate the surrounding stress by this process. In the proposed method, cracks introduce new crack edges in the given input mesh, resulting in newly created faces. Stress is alleviated by distributing the existing stress of the adjacent faces onto the newly created ones. This only alleviates the stress by a small amount and therefore a new parameter is added which controls the stress alleviation of the surrounding faces in the cracking process: the *stress alleviation factor*. This factor is defined in $[0, 1]$, where 0 means no stress remains after the crack and 1 means that the resulting stress is not changed at all. This enables the user to create different crack patterns more easily: a low alleviation factor results in cracks which are further away from each other because the stress in faces adjacent to cracks is lowered, whereas high values do not impose any restrictions on where cracks can occur.

This factor interleaves with the residual weighting value α: a high value of α adds additional stress upon the crack tips, whereas a low stress alleviation value reduces the stress in the adjacent faces of a crack, and with this also the stress at the crack tip.

3.3 User-based Stress Initialization

The stress field can be initialized with several predefined patterns. In this chapter, two interactive stress initialization techniques are proposed which enable the user to directly sketch the stress field onto the mesh or the texture. This allows to incorporate local features within the mesh or the texture. When the user wants to initialize the stress field in form of sketching on top of the mesh, the mouse positions are used to cast rays into the scene to determine which triangle of the mesh has been selected. The user can press a key and move the mouse to insert more than one triangle into a list of triangles, which can be thought of as a painted stroke. Since mouse inputs are checked by most systems multiple times a second, duplicate subsequent triangles have to be avoided. This list is then used to determine the stress in each triangle: at first the direction of the center of a triangle to the next and previous n triangles is determined. The average of these directions is taken to smooth out the overall direction of the stroke. Then a direction orthogonal to the determined direction is used to initialize the stress tensor by taking the outer product of this vector. Taking the outer product of a vector results in a matrix which has the vector as the first eigenvector. The orthogonal vector is used because it is more intuitive that the crack appears in the sketched direction than orthogonal to it and in the cracking component of the algorithm, the orthogonal direction is used to determine the crack direction. The user can define an arbitrary number of strokes which can all be handled in a similar manner. If a triangle lies within more than one stroke, the stress tensor defined by the last overwrites the previous ones. An example stroke on a triangle mesh can be seen in red colour on the right of Figure 20. The triangles which are intersected by the stroke are marked in yellow. Simply taking the direction from subsequent triangles as stress direction results in a jiggled direction as seen in the center of Figure 20. Taking the average of the direction to the n next neighbours in both direction smooths the directions, which can be seen in the right of Figure 20.

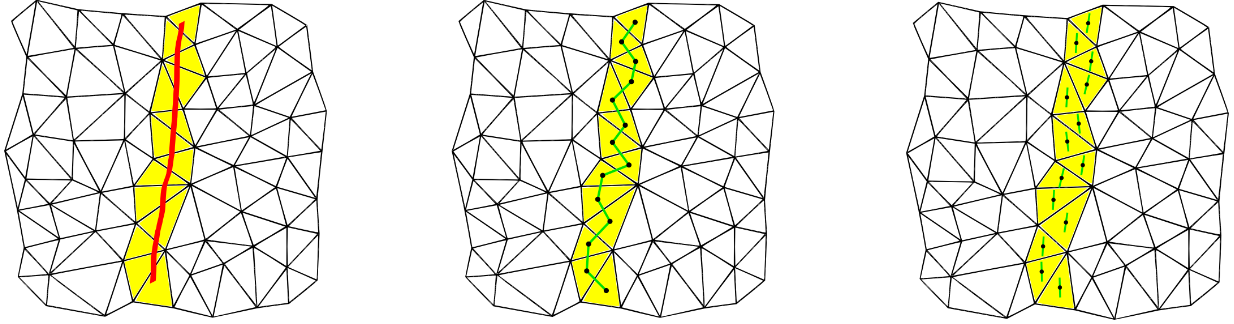


Figure 20: Sketch-based stress initialization. The left image shows an example stroke (red) and the triangles which are intersected by this stroke (yellow). Using the direction between adjacent faces as stress direction results in a jiggled direction, as seen in the center. By calculating the average direction of the n nearest neighbours in both directions on the right, the overall direction gets smoother.

In Figure 21, the user painted one large stroke from top left to bottom right on the mesh. The intersected triangles by this stroke are marked in yellow. After using these triangles and the direction of the stroke to initialize the stress field, a resulting crack pattern can be seen on the right. The rendering technique of the cracks will be described in Section 4.

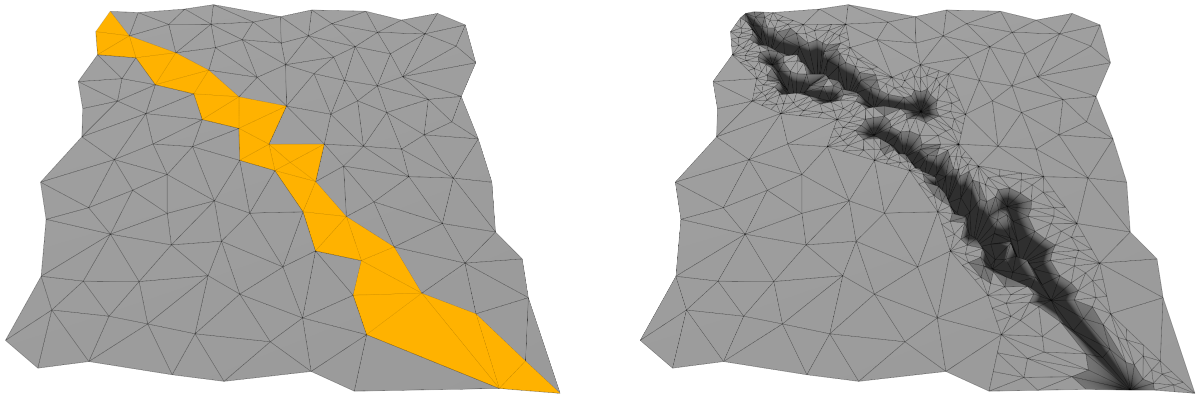


Figure 21: A user-sketched path is used to initialize the stress field. A resulting crack pattern can be seen on the right.

The visual quality of the rendering can be greatly improved by adding a texture to the surface mesh. It provides high frequency data that would hardly be possible on a geometry level. If a texture for the surface mesh exists, the surface cracks should consider the patterns found in the texture. For example when using a bark texture with horizontal cracks, the cracks created by the surface cracking method should also run in the same direction. To

do this, a small tool was developed which renders the given texture on a single quad in an orthogonal projection. The user can sketch paths upon this texture, which can then be exported. Since the texture coordinates are defined in $[0, 1]$, the exported paths also lie within these boundaries. To transfer these into a list of triangles, it is checked if the sketched texture coordinate lies within a triangle. In a naive implementation, every triangle of the mesh has to be checked against each point of the texture sketch. This can be sped up using spatial data structures, such as a quadtree [Finkel and Bentley, 1974]. Multiple texture sketching points can lie within the same triangle which are, similar to the mesh sketching approach, ignored. Figure 22 shows on the left an example bark texture with a vertical crack pattern. The right side shows some example strokes, which sketch this direction at the position of existing cracks. This introduces high stress at the positions of existing crack which have the same direction as the ones from the texture.

Figure 22: Left: bark texture with a vertical crack pattern. Right: stroke sketches defined upon this texture to initialize stress at positions of existing cracks in similar directions.

4 Bark Rendering

The surface cracking method generates crack edges in a mesh. Two methods can be used to render these as described in Section 4: directly render the lines in a different colour or create a visible gap at the cracks by inserting the crack twice and pushing both apart. Both techniques fail to create the appearance of depth and therefore a more realistic and visually pleasing result. In this chapter a more advanced rendering technique is proposed which uses displacement and normal mapping. It uses up-to-date graphics hardware and the latest OpenGL rendering pipeline to achieve compelling results at interactive framerates.

4.1 Normal Mapping

By adding a texture to the surface mesh, the visual quality of the rendering can be greatly improved. However, these textures appear flat due to the lighting calculations. A Blinn-Phong shading model ([Phong, 1975], [Blinn, 1977]) is used to calculate the light intensity I on the mesh. It decomposes light into three parts: An ambient component which is emitted by the object itself, a diffuse component, which depends on the angle between the light source and the orientation of the surface, as well as a specular component which additionally depends on the direction between the observed point and the position of the viewer. Figure 23 shows all vectors necessary for the light computation for a given point: the vector L points towards the light source, N is the normal of the surface at this point and V is the vector towards the viewer (in our case the camera). H is the half-vector of L and V and R is the reflected vector of L at N. The ambient term I_a of the light does not depend on any of these vectors. It describes the light that is directly emitted by the object. The diffuse term I_d is determined by the angle between the light source and the normal of the surface: if the angle is small, the light source shines directly upon the object and therefore the light term should be large. If the angle is large, the ray from the light source only tangentially touches the surface at this point and the light term should be small. This can be encapsulated in the dot product between the light direction L and the surface normal N, the diffuse illumination component I_d.

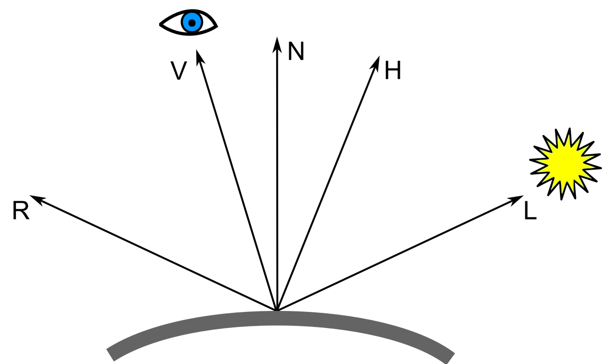


Figure 23: The vectors used for light calculation in the Blinn-Phong model.

$$I_d = max(0, N \cdot L) \tag{10}$$

Since the dot product can be negative if the vectors N and L have an angle larger than 90 degrees, which can happen when the light is behind the object, the maximum of the dot product and zero avoids getting a negative diffuse component. The specular component I_s describes the light that is reflected from the surface into the eye of the viewer. For this, the reflection vector R is determined and, similar to the diffuse component, the dot product between the reflection vector and the view vector determines how close these two rays are to each other. Blinn proposed to use an approximation instead of the reflection vector calculation: the half-vector H between V and L.

$$H = \frac{V + L}{||V + L||} \tag{11}$$

The reflection vector does not have to be computed any longer and instead of calculating the specular component as the dot product of V and R, the dot product between N and H can be used as an approximation:

$$I_s = max(0, N \cdot H)^e \ . \tag{12}$$

The maximum again avoids negative terms and the exponent e controls the appearance of the reflected light on the surface of the object: small values result in larger reflections of the light source and vice versa. In case of only unit length vectors, Equation 12 can be interpreted as a cosine function to the power of an exponent. In Figure 24 the results for different exponents are shown. As described before, smaller exponents create a larger area of influence and vice versa.

Finally, the three light components can be weighted by material-specific factors k_a, k_d, k_s and added up for the final illumination value:

$$I = k_a * I_a + k_d * I_d + k_s * I_s \ . \tag{13}$$

These equations show that the surface normal largely influences the light calculations. Since textures on top of a surface mesh do not change the normals, they appear flat. Normal mapping [Cohen et al., 1998][Cignoni et al., 1998] provides an additional texture containing the normals corresponding to the texture. These can be used to modify the normals of the mesh to incorporate the structure in the texture during the light calculations. The xyz

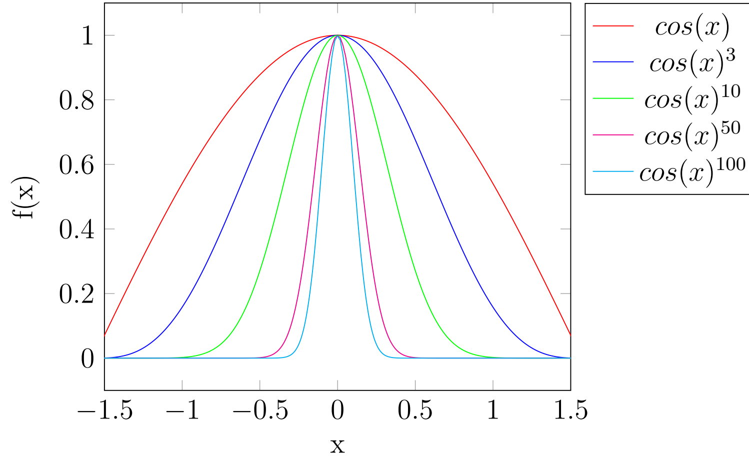


Figure 24: The influence of the exponent on the specular light calculation: small values create a large reflection and vice versa.

values of the normals are stored in the rgb channels of the normal texture. One big problem is that the stored normals, which are defined in the space of each individual triangle (the tangent space), have to be transferred into model space. To do this for a given point on the mesh, three vectors are needed: the normal of the model N, the tangent T of the surface and the cross product of both, the bitangent B. For a given normal, there are an infinite amount of tangents. These have to be chosen in a consistent way to avoid artefacts later on. One standard is to choose the tangent to be in the same direction as the texture coordinates. After having determined the three vectors, they can be combined into the *tbn-matrix*:

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

This matrix transforms normals from tangent space into model space. Normal mapping adds realistic lighting to bumps and dents defined within a texture. In Figure 25 a wall texture is rendered with and without normal mapping. On the left the normal texture can be seen in which the rgb values correspond to the xyz values of the normal. In the center a brick wall texture is rendered with standard Blinn-Phong lighting. On the right the same brick wall is rendered with normal mapping which creates a better perception of depth in the image. A vast amount of free tools exists, which can generate a normal map for a given texture, often using the gradient within the image to calculate the normals. For the proposed rendering method of the cracking model, this technique is used additionally to texturing to create the high frequency details of bark that cannot be created by the cracking model. The crack itself are however rendered with a different technique described in the next section.

Figure 25: Normal mapping applied to a brick wall texture: the left image shows the normal map for a given brick wall texture. The brick wall texture is rendered without (center) and with (right) normal mapping.

4.2 Displacement Mapping

The normal mapping described in the previous section does not change the geometry of the mesh. This produces artefacts at the silhouette of the mesh: while normal mapping creates the impression of depth on a flat surface, the silhouette still remains flat. A different technique is used for the cracks created by the cracking model which modifies the geometry of the mesh. Displacement mapping [Cook, 1984] changes the geometric position of vertices by displacing them, mostly in the direction of the normal. The amount of displacement is determined by a height map.

The proposed method does not use such a height map for the displacement, but directly incorporates the crack edges in the mesh to determine which vertices have to be moved by which amount. Vertices that are close to a crack edge should be displaced by a given distance. For now, let us assume that the distance from the crack edge as well as the amount of displacement is constant for all cracks. Since the displacement mapping is done on the GPU, we have to transport the information about the cracks from CPU to GPU side. To do this, for every vertex two additional information are stored: a flag whether this vertex is a crack vertex, and a crack vertex ID if it is. However, we need the information about the edges on the GPU side as well to determine the distance from a point we want to render to the nearest crack edge. Therefore all adjacent crack vertices that share a crack edge are stored in from of a table where the row number corresponds to the crack vertex ID. An example can be seen in the following table.

# adjacent crack edges	Crack ID	Crack ID	Crack ID
1	4	-	-
3	2	3	7
2	1	5	-
...	...	...	...

The first entry corresponds to the number of adjacent crack edges n and all following n entries correspond to the crack ID of the adjacent crack vertex. This table can be transferred to the GPU in form of a texture. With this information we can calculate for any vertex the minimal distance to a crack edge. If this value is lower than a threshold, which we assume constant for now, the vertex has to be displaced. One of the main problems of displacement mapping is that it requires a large amount of geometry to be able to create granular-enough highlights. This is also a problem when rendering the cracks: additional geometry has to be created close to the cracks. Since OpenGL 4.0, an additional step has been added to the rendering pipeline which makes it possible to automatically tessellate existing geometry. Since we are already able to determine regions which are close to a crack edge, we can use this tessellation step to create further geometry in this region. Figure 26 shows the tessellation applied on a mesh, where the faces adjacent to the crack (marked red) are subdivided into smaller faces.

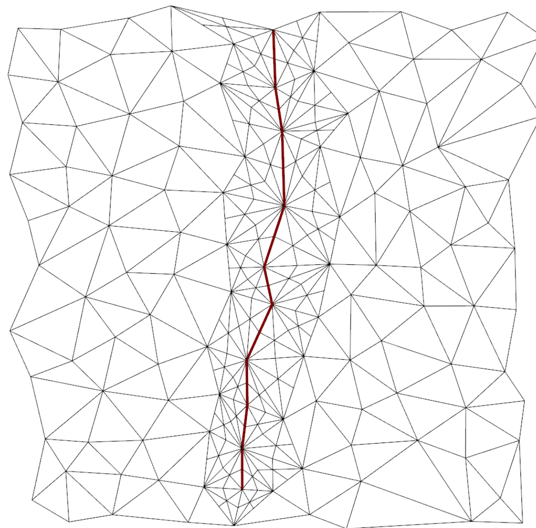


Figure 26: The faces around a crack (marked red) are tessellated with the built-in OpenGL 4.0 functionality.

Having enough geometry in close proximity of the crack edges allows to create highly detailed cracks by displacing the newly created vertices. Keeping the displacement value constant as assumed up to now results in a flat and unrealistic appearance of the crack.

Looking at bark cracks in nature shows that the depth of the cracks increases closer to the center of the crack, creating a V-shaped profile. Longer cracks also increase in width from their starting point and decrease afterwards to an end point. The depth of the crack depends on the crack width: wider cracks are usually deeper. To create cracks that closer resemble those appearing in nature, the first thing that has to be determined for every crack vertex is the width of the crack at this position. To do this, the crack edges are divided into connected components, as seen with different colour coding in Figure 27.

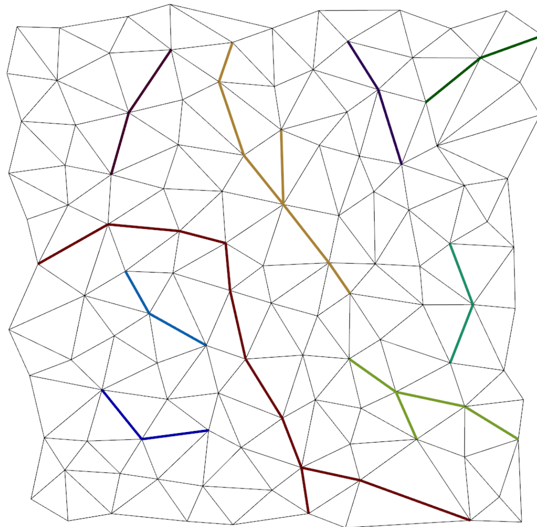
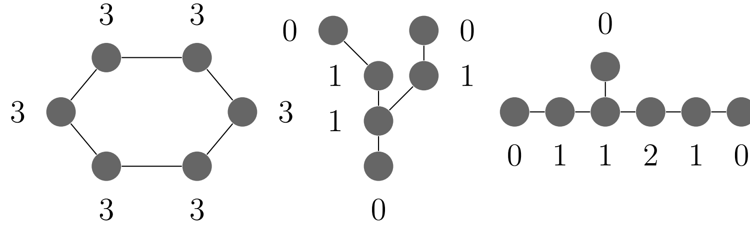


Figure 27: Crack edges divided into different connected components (color coding).

These can be used to determine where in the crack the vertex lies. Closer to outer vertices, which have a degree of one with regard to the crack edges, the crack width should be small and increase the further away the vertex is from these. To determine this width, Algorithm 1 has been used. Starting from the crack vertex v, we want to determine the width of the crack at the position. This is done by doing a breadth first search in the connected component the vertex lies in. As soon as a vertex with degree equal one is found, the algorithm returns the distance to this vertex. The result can be thought of as minimum distance to an outer vertex of the connected component. In case of a circular connected component, no vertex has a degree of one. The algorithm then stops when no new vertices are found. Three example connected components with the calculated distance values are displayed.

Algorithm 1: Determine crack width

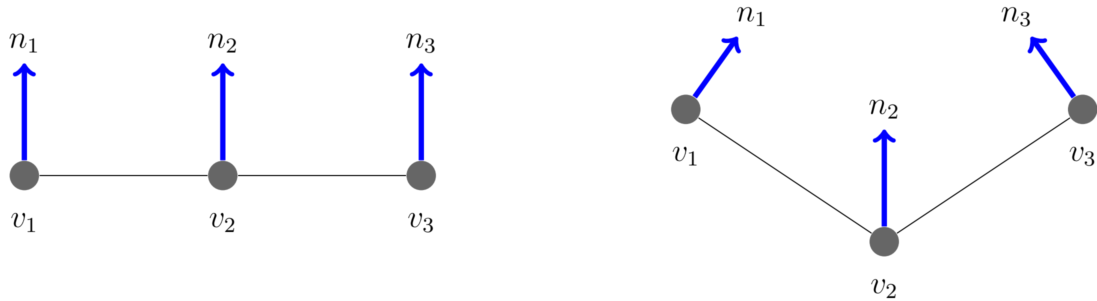
```
Data: Vertex v, Conntected component c
Result: Width of the crack
Float distance = 0;
Queue current;
Push v into current;
while current not empty do
    Queue next;
    foreach Vertex in current do
        if degree of vertex == 1 then
            return distance;
        else
            if vertex not yet seen then
                add vertices in c adjacent to vertex into next;
            end
        end
    end
    current = next;
end
return distance;
```



These values are used to determine the width of a crack at the position of a crack vertex. The depth of the crack is created by the displacement value. As stated before, the depth should increase from the sides of the crack and be the deepest in the center of the crack creating a V-shaped profile. For any given vertex that has at most the maximum distance d_{max} to the crack edge, the distance d is compared to the maximum distance: the more the distance goes towards the maximum distance, the lower the displacement value should be. These are the outer sides of the crack edge. If the distance to the crack edges goes towards zero, it comes closer to the center of the crack. These regions are displaced the most. The displacement value ψ can be determined as:

$$\psi = k \cdot \frac{d_{max} - d}{d_{max}} \ ,$$

where k is a user specified value which defines the maximum displacement depth. For every displaced vertex the normal has to be recalculated since the surface going through this vertex has been changed as well. This also has to be done for every vertex that shares a face with a displaced vertex. The following example illustrates the situation before and after vertex v_2 has been displaced.



To determine the new normals at the vertices, as a first step the normals of the faces adjacent to each vertex are calculated. Assuming the faces are triangles, the new normal is the cross product of the two spanning vectors of the triangle. The average of these adjacent face normals then defines the new vertex normal. The deeper the crack, the more light is obstructed from illuminating it. To incorporate this, the lighting calculated in the deeper regions of the cracks is dampened by a factor depending on the crack depth. This factor can be stored in the length of the normals: smaller normals result in a smaller value during the lighting calculations. Therefore larger displacement values are coupled with smaller normals in the normal recalculation step. Cracks created with this technique have the general appearance of bark cracks in nature, but they appear very uniform. Noise can be used to increase the realism and the variety of crack appearances. The crack width and the displacement value are two examples where noise can be applied. An example for a crack rendered with the described technique can be seen in Figure 28.

Throughout the presentation of this method multiple parameters that control the rendering of the cracks have been introduced: the crack width determines the distance vertices can have to an existing crack edge to be regarded as displacement vertices. The depth of the cracks can be controlled via the depth factor in the displacement value calculation and the darkening of the deeper crack regions can be controlled with a depth darkening factor. This gives the user full control over the real-time rendering and makes this technique suitable for an artistic tool. Drawbacks of this method are mainly caused by the underlying displacement mapping technique. If a very small crack edge is wanted, the mesh has to be tessellated very highly which might result in a poor performance due to the large amount of created geometry. Also self-intersection artefacts are possible when the displaced vertex is moved through the other side of the objects which might often be the case with thin objects.

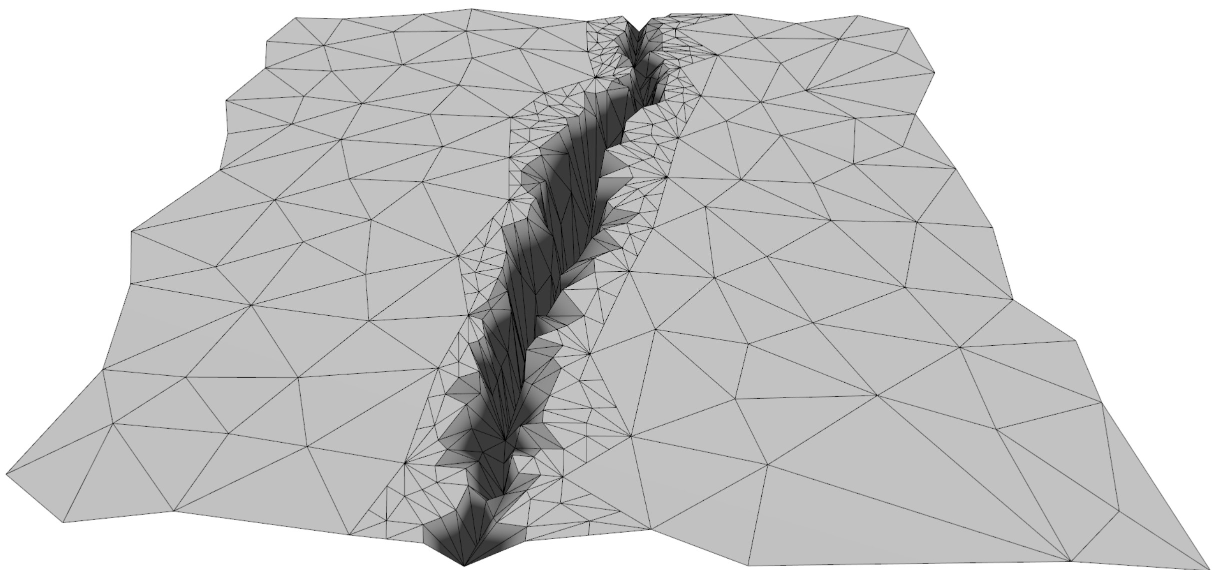


Figure 28: A crack rendered with the proposed method incorporating displacement mapping and tessellation. Triangles close to the crack edge are tessellated into smaller triangles. These are displaced depending on the distance to the crack, resulting in a V-shaped profile.

5 Evolving Meshes

In the previous section, cracks occur due to stress in a surface mesh. The stress is mainly caused by the growth of the tree. In this section, a method for mesh evolution is presented. It allows to define the growth of objects and is also capable of growing objects around obstacles. The surface cracking method can be coupled with the evolving meshes to create more realistic results and an even more faithful simulation of the natural process that leads to cracks in bark. This section is structured in the following way: Section 5.1 gives a short introduction into the field of deformable interfaces. Section 5.2 shows the most important related work in this field and Section 5.3 presents the method used for mesh evolution (for a more complete introduction see [Nielsen, 2006]). In Sections 5.3.1 to 5.3.3, different growth models are introduced and in Section 5.3.4 an interactive technique is presented for sketching the growth on a surface mesh, which is similar to the technique used for the stress field initialization of the cracking algorithm. Lastly, Section 5.4 describes the coupling of the surface cracking algorithm and the mesh evolution.

5.1 Introduction

Many applications deal with the problem of tracking a changing object over time. The border of an object in this context is called its interface. An important concept in this field is the assumption that the interface will change over time. This also includes that interfaces may split up into multiple parts or merge into fewer. In fluid dynamics it is often necessary to track the interfaces of different fluids when they are in motion. The processes in car engines can be efficiently simulated with such techniques. They can also be used for segmentation or object tracking in image processing and in 3D modelling where the tracked interface is the surface of the object.

5.2 Related Work

Methods for tracking deformable interfaces can be divided into *explicit (Lagrangian)* and *implicit (Eulerian)* approaches. The changes applied to the interfaces are given in form of an underlying velocity field u. Explicit methods parametrize the interface of objects and apply the velocity field onto the points of the interface p:

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = \mathbf{u}(\mathbf{p}) \ . \tag{14}$$

With a non-uniform velocity field, the topology of the interfaces changes, which makes a new parametrization necessary. These changes can also result in self-intersections of the

interface, which make a reparametrization obligatory along with changes in the interface [Glimm et al., 1995]. These problems do not occur in implicit methods. The most well-known method in this field is the *Level Set Method (LSM)* [Osher and Fedkiw, 2003], which is an implicit surface representation of the interface, assuming that all implicit surfaces might be disconnected, but have to be closed. Implicit surface representations define a surface as an isocontour of a scalar function which maps the parameters (2 for curves, 3 for volumes) to a scalar value (called a surface embedding $\phi : \mathbb{R}^3 \to \mathbb{R}$), which is called the isovalue k. An isosurface contains the set of points in $\mathbb{R}^3$ defined by $\{x|\phi(x) = k\}$. With this for example it is easy to describe a circle with radius r as $\phi(x, y) = x^2 + y^2 - r^2 = 0$. An implicit surface representation does not specify which points lie on a surface. Instead it allows to evaluate for a given point whether this point lies on the surface or not. Given a constant function, all points that lie on the same isosurface create the *level set*. It is often assumed that the tracked interface is defined by the zero-isocontour and that the surface embedding maps points from the interior and exterior regions to values with different signs. This property is the reason behind the first two advantages of level sets over explicit approaches: determining whether a point lies inside or outside a surface can be simply done by evaluating the surface embedding once. Self-intersections are not possible, since the sign of the scalar result of the embedding function can be either positive or negative, not both at once. In computer science, level set surfaces are usually not represented analytically. This is mainly due to the fact that for most given interfaces no direct analytical description is available. Instead, the surface embedding function is sampled on a grid. During the interface deformations, the grid stays static at the initial position. The interfaces moves by modifications of the scalar values of the embedding function. The most commonly used function is the signed distance function. It assigns to each point the minimal distance to the surface multiplied with $+1$ if in the exterior and by -1 if in the interior. The circle equation seen above is an example for such a function. The time dependent velocity field u assigns to a given point a velocity at time t: $u(\mathbf{x}, t) : \mathbb{R}^3 \to \mathbb{R}^3$. This movement can be described by $\mathbf{x}(t) = (x(t), y(t), z(t))$. The dynamics of the level sets can be achieved by coupling this expression with the surface embedding ϕ: $\phi(\mathbf{x}(t), t) = 0$. Applying differentiation to this time-dependent surface embedding results in the *level set equation*:

$$\frac{\partial \phi}{\partial t} + \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} \cdot \nabla \phi = 0 \ . \tag{15}$$

The disadvantages of the LSM are mainly caused by the explicit sampling of the embedding.

- Computational costs and storage requirements are proportional to the size of the sampled gird.
- Interface tracking can only happen inside the initially defined grid.
- Fixed resolution of the grid can create sampling artefacts.

- Numerical diffusion for features near the sampling rate.
- Isosurface has to be determined, i.e., by marching cubes [Lorensen and Cline, 1987].

Because of the widespread use of the LSM, many other papers have been published that try to overcome some of these disadvantages. The computational costs and storage requirements have been tackled by a narrow band level set method which only solves the method in close vicinity of the interface [Chopp, 1993]. The same problem has been addressed using an octree-based approach that has a higher resolution close to the interface and vice versa [Droske et al., 2001][Losasso et al., 2004][Losasso et al., 2006]. An increased numerical stability is the goal of an approach, which additionally introduces particles into the otherwise Eulerian approach: these particles are used in regions of high curvature and under-resolved regions to ensure greater accuracy [Enright et al., 2002]. A grid-resizing strategy is employed to resolve the fixed resolution [Adalsteinsson and Sethian, 1995], and also an approach which is capable of extending the domain is presented [Bridson, 2003].

5.3 Deformable Simpicial Complex Method

The *Deformable Simplicial Complex (DSC)* method [Misztal and Bærentzen, 2012] is a Lagrangian approach that, similar to LSM, avoids self-intersections and therefore costly mesh operations. In the DSC method the interface is represented explicitly as a set of faces of simplices one dimensional higher, which either belong to the object or the exterior. In the 2D case, the domain is divided into triangles, and in the 3D case into tetrahedra. Since the underlying structure is a tetrahedral mesh, the triangle meshes used up to now have to be converted. For this purpose we found the *TetGen*[1] library to be fitting. The interface is deformed by moving the vertices. This avoids the numerical diffusion problem of the Eulerian approaches like the LSM. It is always explicitly present in the mesh and does not have to be computed like in the LSM. When the interface moves, the mesh is updated automatically. This is usually done along the surface normal n_i, depending on a time dependent speed function $s(x_i)$.

$$\dot{x_i} = s(x_i, t) \cdot n_i \tag{16}$$

[1]http://wias-berlin.de/software/tetgen/

The DSC method overcomes the self-intersection problem present in LSM approaches by using the interior and exterior mesh to check if a movement of a vertex would result in an intersection between these two. If so, the vertex is only moved as far as it is possible without such an intersection. The surface triangle mesh always lies within the intersection of both meshes and the method allows to specify whether two interfaces merge or stay separated when they meet. Tetrahedrons are also locally remeshed when the movement creates degenerated versions. This allows subsequent movements to come closer to their propagation destination points. Smoothing and adaptive refinement can improve the mesh quality. In Figure 29, two tetrahedral meshes are propagated into the direction of each other. When the two boundaries (interfaces) meet, the propagation will be stopped while maintaining the boundary between the two. This shows how the method resolves geometric and topological problems that can occur when two objects or an object and an obstacle intersect. Obstacle in this context describes an object that does neither evolve nor move. This would not be possible with a LSM approach since the interfaces would merge, which is unwanted in case of tree growth.

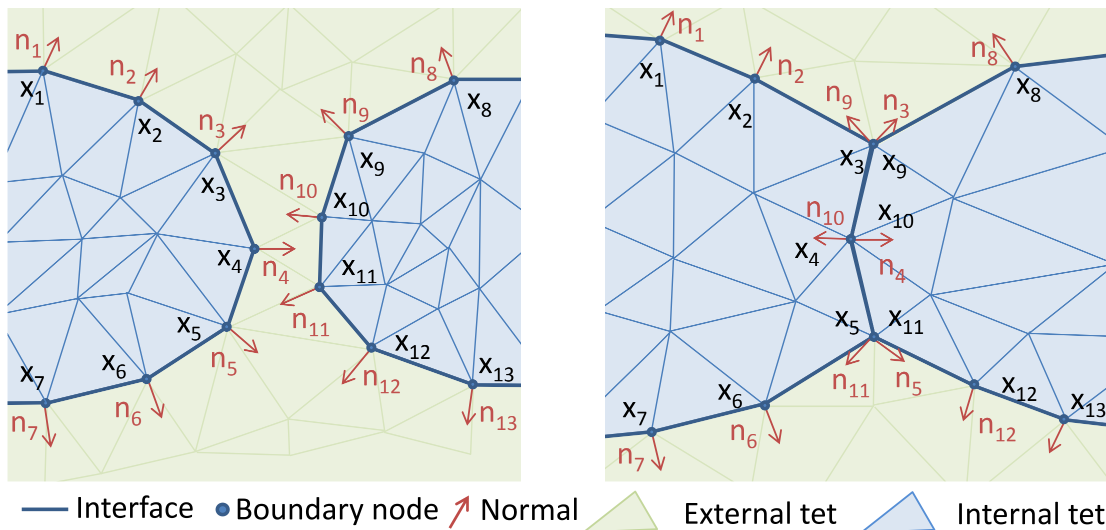


Figure 29: Two tetrahedral meshes are propagated into the direction of each other. When the interfaces meet, the propagation of the vertices at these positions stops, leaving the boundary intact (Image: Purdue University).

After converting a triangle mesh into a tetrahedral mesh, only the normal vector and a growth function which defines for each vertex a scalar growth value are needed. Since the normals can be determined by the tetrahedral mesh, only the growth function has to be defined. The growth is calculated using an Euler integration for each vertex. The aim of this method is the simulation of tree bark development and therefore the applied growth functions have to conform with the growth of a tree. Three growth functions are proposed which are biologically-motivated and also an user-defined growth function provided by a brush-like interface is suggested in Section 5.3.4.

5.3.1 Allometric Growth

This is a data-driven approach which models tree growth based on field measurements. A single-parameter Weibull distribution function is used to determine cambial growth and therefore the circumference of the tree [Landsberg and Sands, 2011]:

$$s(t) = ct^{c-1}\, e^{-t^c} \ , \tag{17}$$

where t is the time of the growth and the parameter c depends on the tree species. Figure 30 shows the function for different values of c. It shows increasing growth rates at the beginning and declining afterwards. Depending on c, the maximum growth rate is reached faster for a low value of c and vice versa.

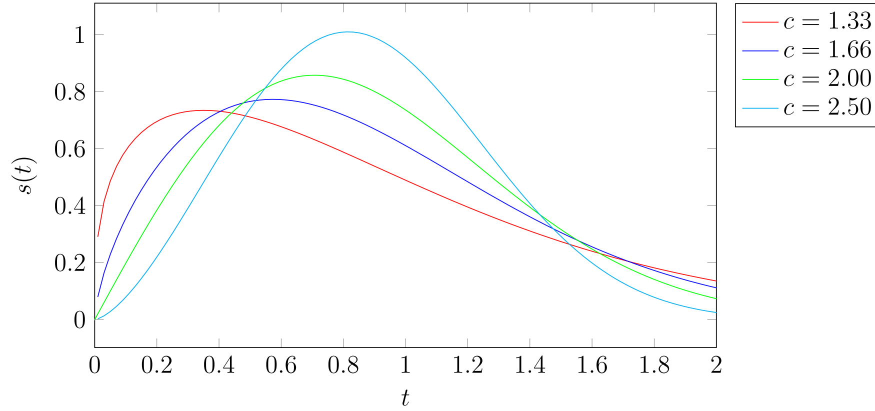


Figure 30: Different values for c in Equation 17.

5.3.2 Structural Growth

When part of a tree bark is subject to mechanical stress, reaction wood is formed to support the structure of the tree and bring parts of it in more optimal positions. Reaction wood can be divided into compression and tension wood. The presence of reaction wood is approximated through an eccentricity parameter e:

$$s_e(x,t) = (1 + e \cdot n(x) \cdot y) \cdot s(t) \ , \tag{18}$$

where y is a unit vector in vertical up direction. Compression wood is modelled by negative eccentricity, whereas tension wood is modelled by positive eccentricity.

5.3.3 Reversed Growth

To determine annual growth rings, an apical growth model would be necessary. However, we only have a tree model at maturity. In order to simulate tree growth rings, the simulation is done backwards in time through successive erosion operations by negating the speed function. This growth includes a small factor depending on the curvature to make sure that the tree erodes to sapling diameters without changing the topology from the mature tree. Grayson's theorem states that a simple planar curve erodes under its curvature smoothly to a point [Sethian, 1999], so its extrusion into a generalized cylinder should erode to the spine which was determined by the sapling. Different time steps can be used to create meshes at different erosion states. Inserting these meshes into a singular mesh in a darker colour than wood produces growth rings when intersected with a plane in wood colour.

5.3.4 User-defined Growth Sketching

In addition to the growth functions mentioned above, a more interactive growth definition is proposed. Interactive sketching on the object surface allows the user to have more control over even smaller regions of the growth. It works similar to the proposed method to sketch the stress initialization in the cracking algorithm. Ray-casting is used to determine the selected triangles on which a scalar value is added. A brush size can be defined to modify a larger area of the object surface at once with a Gaussian filter to give more weight to the triangles closer to the mouse position. The scalar value is then interpolated at each vertex with its surrounding triangles and used as the growth value. An example for a sketched growth function can be seen in Figure 31. The first image shows the surface mesh, the second image shows a regular growth function depicted as the equal length of the surface normals. In the third picture, the user has defined more growth in regions of the chest, ears, and on top of the head of the cat. The resulting evolved mesh can be seen in the last picture.

5.3.5 Growth Around Obstacles

In nature it can often be observed that trees grow around obstacles like fences, walls, or other trees. The growth is not inhibited by the obstruction and the tree eventually absorbs the obstacle into its interior by growing around it. The obstructions can be modelled as tetrahedral meshes inside the external mesh. It is possible to keep these meshes static so that the DSC front propagation does not change them. By embedding the obstacles in the external mesh, a front-obstacle collision detection step is avoided. Because the obstacle mesh is static, other fronts cannot penetrate it. Parts of the front close to the obstruction can continue their growth. The normals of the front neighbouring the obstruction will turn

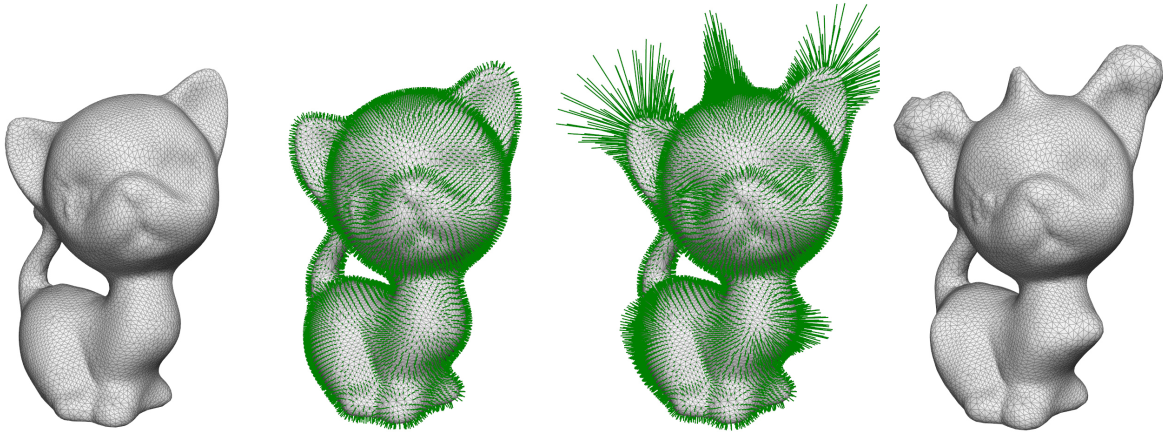


Figure 31: The amount of growth of the growth function is displayed as the length of the normal vectors (green). Regions with more growth at the chest, the ears, and on top of the head of the cat have been defined by the user. The resulting evolved mesh can be seen on the right (Image: Purdue University).

to face the obstruction and eventually surround it. The front will then move towards itself. Since the DSC method can maintain the fronts after self-intersection, a realistic crease is created. Two such creases due to self-intersection can be seen in Figure 29 between vertices x_3/x_9 and x_5/x_{11}.

5.4 Evolving Meshes with Surface Cracking

The presented method for mesh evolution is used to simulate the growth of wood-like structures. In the growing process an increasing stress is created in the bark which results in the formation of cracks. This section describes the cracking model applied to evolving meshes and the resulting challenges. For the mesh evolution with the DSC method, the input triangle mesh is transformed into an interior and exterior tetrahedral mesh. A growth function is applied to propagate the vertices of this mesh. At any point in time, we can re-transform the tetrahedral mesh into a triangle mesh to become a evolved triangle mesh which can be used for the cracking model. Duration and time steps of the evolution can be chosen freely by the user. In addition to different vertex positions, the evolved mesh can also have newly added or removed vertices and edge swapping, due to the local remeshing process that is necessary to have a smooth front propagation during the evolution process. We assume that the time step between two states of the evolved meshes is small enough to have only some local changes. This enables us to track which vertices are newly added and which vertices have moved in which direction to a new position. We use two techniques to track vertices: the first technique is to label vertices within the DSC library with an

additional ID attribute. When a vertex has the same ID before and after the evolution, we can be sure that the vertices are the same. Since remeshing can remove or insert vertices it is possible that a vertex has no previously assigned ID. If it is close enough to a previously existing vertex for which no other vertex with its corresponding ID has been found, our second technique assigns this vertex the same ID. If the time step is too large, we can simply make it smaller and use more time steps. For the rendering of the meshes, we want to incorporate previously described techniques such as texturing.

5.4.1 Texture Coordinate Propagation

Since the texture coordinates are only given for the initial mesh, they have to be determined for an evolved mesh. There is currently no functionality for this purpose in the DSC method. In case of a vertex being removed, its texture coordinates are removed as well. When a new vertex is inserted, the texture coordinates of the neighbourhood are used to interpolate new texture coordinates. This works well under the assumption that only small changes happen between time steps: at least some neighbouring vertices already existed in the previous time step and therefore have texture coordinates. When a vertex moves, the movement can be decomposed into normal and tangential components. The normal component does not change the texture coordinates, whereas tangential movement requires texture movement. This can be determined similar to new vertices from the local neighbourhood. An example for an evolved mesh at two different time steps with a checkerboard texture can be seen in Figure 32. Assuming two triangle meshes at different time steps t and $t+1$ of the

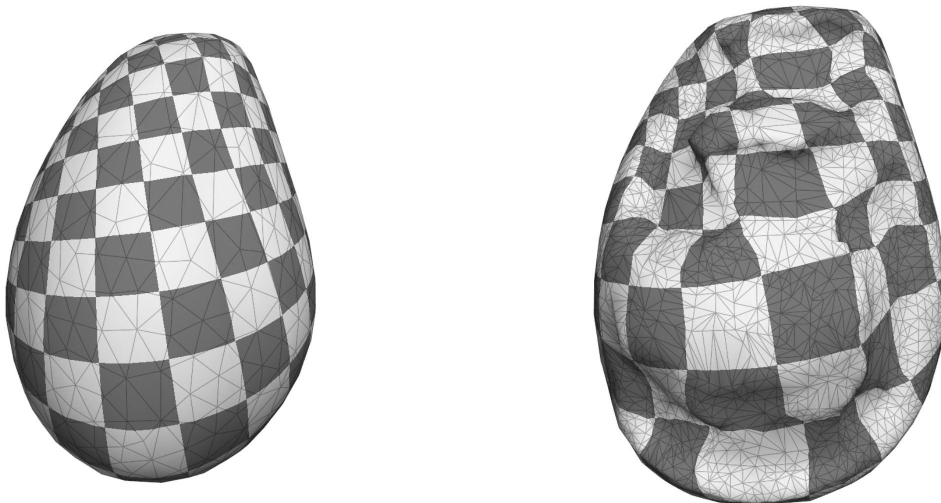


Figure 32: Texture coordinates are propagated on an evolving mesh from one time step to the next via interpolation (Image: Purdue University).

evolution process, we do not want to start the cracking process from anew at time $t + 1$. Since the mesh has changed, it is not guaranteed to get the same cracks as at time t. If the cracked evolved meshes are to be used for an animation, this is necessary to avoid flickering artefacts.

5.4.2 Transfer of Cracks

To avoid artefacts when cracking an evolving mesh, the cracks from time step t have to be transferred to time step $t + 1$ before starting the cracking on the mesh at $t + 1$. We previously stated how we track vertices between time steps. The easiest case to transfer a crack edge is if both vertices of this edge can be tracked in the evolved mesh. If these vertices lie within a common triangle, the edge connecting them is simply marked as a crack edge. Local remeshing operations can change the local neighbourhood dramatically. If two vertices sharing a crack edge at time step t do not lie within the same triangle at $t+1$, they cannot be connected without invalidating the triangulation. Because of the small time between two time steps it is assumed that these vertices lie at most within adjacent triangles. These triangles are then remeshed in order to create a crack edge connecting these two vertices. If at least one vertex of a crack edge cannot be tracked or they do not lie at least in adjacent triangles, they are not transferred between the time steps. The user has to choose a smaller time step so that the transfer falls in one of the previously mentioned cases. The time step also depends on the growth that is used in the evolution process: a fast growing mesh requires a smaller time step than a slow growing one.

5.4.3 Stress Field Transfer

Cracks alleviate stress in the surrounding areas. This largely influences where cracks are created in successive time steps. To ensure that cracks transferred from time step t to $t + 1$ behave correctly, the underlying stress field has to be transferred as well. The stress field is defined as stress tensor per triangle at time step t. To transfer these stress tensors to a different mesh at $t + 1$, a similar approach to the crack edge transfer is taken. In the simplest case, each vertex of a triangle can be tracked to the evolved mesh. The stress tensor is then simply used for the new triangle. If this is not the case, no stress is initialized in the triangle from the evolved mesh. Since we assumed that only small local changes happen between time steps, the triangles with no initialized stress occur only in a few cases. Stress can then be interpolated between adjacent triangles that have been initialized via the tracking process. The interpolation can for example be weighted by the distance between the center of the triangles.

5.4.4 Stress by Growth

The growth of a tree introduces stress in the bark due to the increasing circumference. This stress can create cracks in the bark of the tree. We want to simulate this behaviour when the mesh grows. To do this, we initialize stress on the initial model of the tree, either with predefined stress patterns, or sketched by a user either on the mesh itself or on the texture. When the mesh grows, we transfer the stress from one time step to the next as described previously. To incorporate the stress growth in the surface, we increase the transferred stress by a user defined value. Assuming similar growth for all regions, this factor is constant. However, the stress should be increased more in regions of larger growth. The displacement value is used to determine regions with larger growth and therefore a larger increase in stress. Additionally, curvature could be used to introduce additional stress in regions with a large distortion, because these regions would naturally be more prone to cracks.

5.4.5 Technical Overview

From the implementation side, the different time steps of the mesh are stored as files. In a first step, a given input triangle mesh is transferred into a tetrahedral mesh and then evolved by a growth function, which is either predefined or interactively produced by a user. The time step in which the meshes are stored can be freely chosen. In a second step, the surface cracking method is applied to these meshes. A system based on keyframes is developed with which the user can easily define cracking parameters for the meshes at some of the time steps. Parameters are interpolated for the meshes in between the keyframes. The previously described technique for crack and stress transfer is automatically applied between subsequent meshes.

6 Results

This section contains a variety of result images from the presented surface cracking method on different models. Also examples for time-lapse animation and evolving meshes are given. The parameters used for the renderings are given in Table 1. Figures 33 and 34 compare a tree trunk model with different textures and normal mapping with and without the cracking method applied. The cracks increase the realism and the objects appear less synthetic. Another effect due to the cracks is the visual ageing of the tree: older trees usually have more cracks than younger trees. The synthetically achieved results are compared in a close-up view to real world photographs in Figure 35. For this, cracks have been manually removed (using the context-sensitive filling technique in Photoshop [Barnes et al., 2009]) and used as background for the synthetic cracks.

Figure 33: Tree trunk (palm texture) without and with cracks.

Figure 34: Tree trunk without and with cracks. Birch texture top, pine bottom.

Figure 35: Tree bark comparison of different species between photograph (left) and synthetic result (right). Species from top to bottom: palm, birch, pine.

Figures 36, 37, 38, 39, and 40 show different results for static meshes and different stress patterns used. They show that merely with the change of some parameters, quite different results can be achieved, which resemble those of barks of various tree species. A result for a non-static mesh can be seen in Figure 41 and another result for the growth around an obstacle can be seen in Figure 42, which shows the mesh at two different stages of the evolution: one before the tree mesh grows around the fence obstacle and one after. The cracking parameters for the two steps are separately listed in Table 1. Time-lapse style animations for two different models can be seen in Figure 43 and 44. The parameters in Table 1 are for the full animation. The total iterations parameter is interpolated linearly between the different time steps which have an equal distance.

Figure 36: Cracked bunny. 4378 triangles, 7s computation time.

Figure 37: Cracked mug. 12845 triangles, 33s computation time.

Figure 38: Cracked knot (13110 triangles, 111s computation time). Cracked gargoyle (50000 triangles, 206s computation time).

Figure 39: Cracked vase. 25941 triangles, 45s computation time.

Figure 40: Cracked neptune statue (56112 triangles, 270s computation time). Cracked mouse figure (41602 triangles, 610s computation time).

Figure 41: Bunny before and after mesh evolution. The growth has been defined in the direction of the surface normals.

Figure 42: Tree growing around fence. The DSC method stops the interface propagation when an obstacle is met.

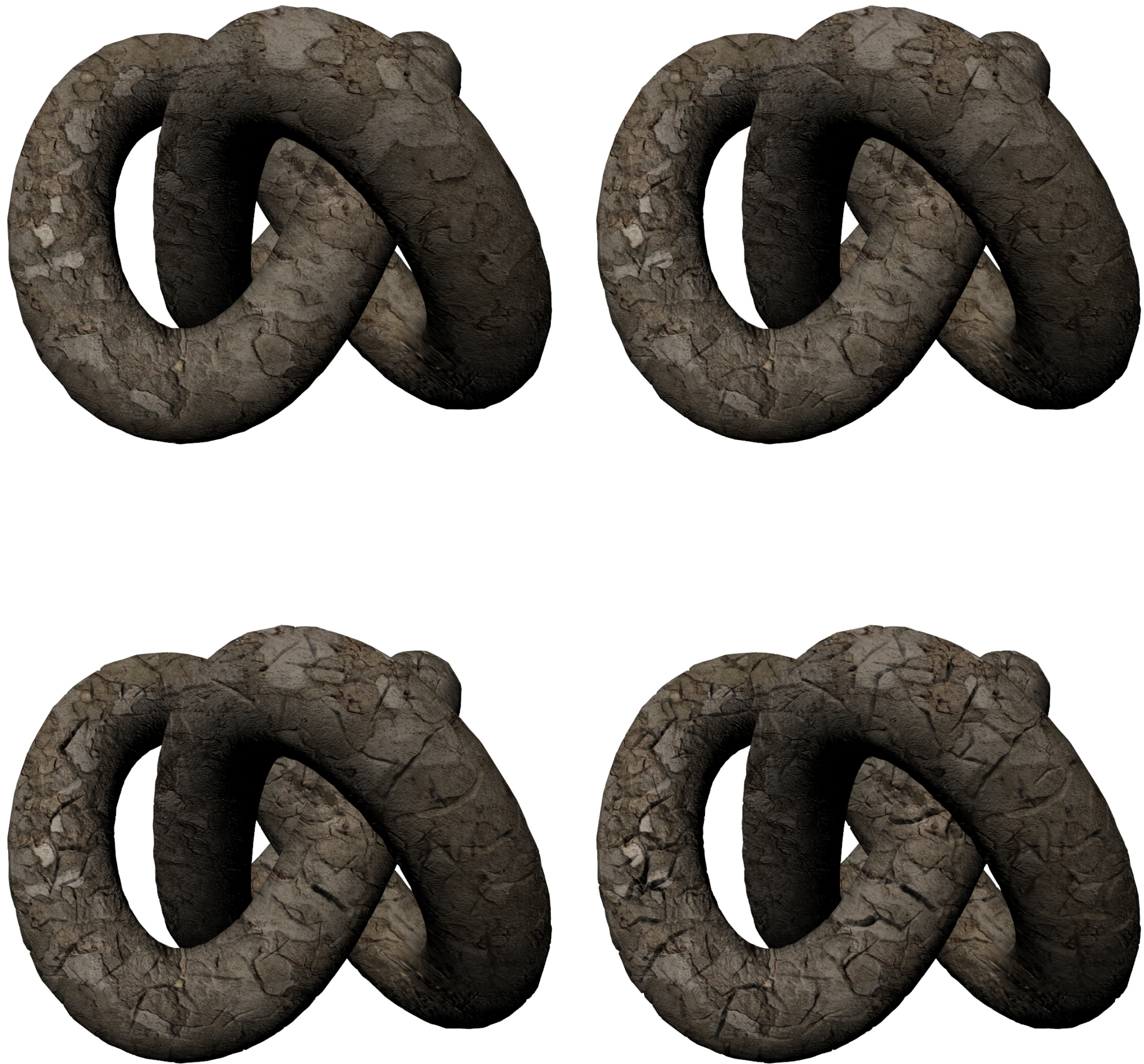


Figure 43: Time-lapse style animation: knot.

Figure 44: Time-lapse style animation: cup.

Mesh	**Triangles**	**TI**	**CPI**	**RPI**	**alpha**	**SA**	**RR** (π)	**Pattern**	**Time** (s)
Neptune	56112	200	10	3	0.3	0.5	0.0	horizontal	270
Vase	25941	100	8	5	0.1	0.0	0.2	horizontal	45
Mug	12845	120	10	1	0.1	0.25	0.0	uniform tension	33
Gargoyle	50000	175	10	5	0.2	0.0	0.1	horizontal	206
Knot	13110	125	15	5	0.2	0.0	0.1	uniform tension	79
Bunny	4378	75	8	3	0.1	0.3	0.0	uniform tension	7
Mouse	41602	250	15	2	0.0	0.1	0.0	uniform tension	610
Tree trunk (birch)	7852	50	5	1	0.0	0.0	0.0	user defined	4
Tree trunk (pine)	7852	120	8	5	0.15	0.25	0.0	user defined	22
Tree trunk (palm)	7852	100	8	2	0.3	0.0	0.0	user defined	15
Knot animation	13110	150	15	1	0.15	0.0	0.0	uniform tension	111
Mug animation	12845	120	12	1	0.1	0.25	0.0	uniform tension	44
Tree & fence (before)	34074	150	10	5	0.3	0.25	0.0	user defined	124
Tree & fence (after)	48698	300	10	5	0.3	0.25	0.0	user defined	450
Bunny Growth	43372	100	25	5	0.5	0.25	2.0	vertical	390

Table 1: Simulation Parameters: Total iterations (TI), Cracks per iteration (CPI), Relaxations per iteration (RPI), Residual value (alpha), Stress alleviation factor (SA), Random stress tensor rotation (RR), used stress pattern and computation time.

7 Conclusion

The presented method was implemented in C++ and OpenGL was used for the rendering. Computation times of the cracking algorithm mainly depended on the mesh size and the number of iterations of the algorithm, with the relaxation step taking up most of the time. Since many steps like the relaxation and force calculations are independent for each face/vertex, they have been parallelized on the CPU. *Qt*'s[2] thread framework has been used to achieve this. Since the mesh grows when cracks are added, the later iterations of the algorithm take up more time. The system specifications which have been used to determine the computation times are an Intel i7-4770K CPU, a Nvidia GTX 780 with 3 GB VRAM and 16 GB RAM. A resolution of 1920×1200 pixels has been used for the rendering. This system was able to render all results at interactive frame rates (> 25 frames per second) and the main bottleneck was the tessellation of the area around the cracks which creates a large amount of geometry compared to the rest of the model.

With this method it is possible to create a broad variety of tree bark crack patterns that resemble those appearing in nature. Some of the results have been compared to real world photographs to demonstrate the amount of realism that can be achieved. The user has a large set of parameters to control the appearance of these cracks. The user control and the possibility to interactively set the stress and the growth field make it especially viable for an artistic tool. Some of the divergent results which can be created have been seen in Section 6. The technique has also been extended to work on a set of evolving meshes. Cracks created with this method are consistently transferred between subsequent meshes and the growth process introduces new stress in the model, which is then used by the cracking algorithm.

One drawback of the method is the dependency on the initial mesh quality. For a regular mesh, the resulting crack patterns appear regular as well. Also the parameters are somewhat dependent on the mesh they work on: for some crack patterns and a given parameter set the initial amount of geometry existent in the mesh might not be enough. Using a mesh with more initial geometry however requires adapting the parameters to this larger mesh. The total iterations for instance have to be increased so that the crack patterns do not appear smaller than before. The displacement mapping can create artefacts when displacing vertices due to self-intersections by pushing them through the other side of the mesh.

The main contribution of this work is the developed cracking model specialized for the creation of surfaces with the appearance of tree bark, which adapts an existing method. Also the interactive sketching of stress and the rendering have to the best of our knowledge not yet been done in this way. A second contribution is the coupling of the method with a mesh evolution technique.

[2] `qt-project.org`

8 Future Work

Crack Simulation on GPU

The force calculation and relaxation steps of the surface cracking algorithm work independently on different faces or vertices. Currently these steps are parallelized on the CPU. To achieve an even greater amount of parallelism, these steps could be transferred onto the GPU. This would require to adapt some of the structures currently used by the CPU implementation to be suitable for the GPU and frameworks like CUDA[3].

Limited Displacement Depth

If the displacement mapping moves vertices by a too large amount, the vertex can penetrate the back side of the mesh, creating unwanted artefacts. Especially objects which are very thin or have very thin parts suffer from these artefacts. The self-intersection limits the maximum amount of displacement that can be used for the whole rendering of the cracks because the displacement depth is determined by the algorithm according to the size of the crack. The problem could be solved with a ray tracing approach: a ray from the vertex that has to be displaced is cast in direction of the displacement. The distance from the vertex to the next point hit by the ray limits the displacement value and thereby avoids self-intersections within the mesh.

Coupling with Volumetric Crack Approach

The presented cracking model aims to simulate cracks appearing on the bark of a tree. The cracks are approximated by modelling the growing surface of the bark. However, other cracks can also appear in the heartwood (the wood inside the cambial layer) and propagate to the outside. Since the process is happening inside the volume of the tree, the surface cracking model does not suffice. A volumetric crack approach could be incorporated in addition to the surface cracking model, specialized in these kinds of cracks. The volumetric predecessor paper from the paper of which the surface cracking method was adapted would be a possible candidate for this.

[3] `www.nvidia.com/object/cuda_home_new.html`

List of Figures

1 Different materials which define their appearance by the surface structure: Dried mud (image Frank Vincentz), broken concrete (image Dmitriy Chugai) and rusty metal (image `fantasystock.deviantart.com`). 1
2 Tree bark is mostly defined from its surface structure. It greatly varies between different species. The example shown are from a palm, a young birch and a pine. 2
3 Top: segmented texton channels. Bottom: input image left, merged channels for fracture middle, for lenticels right. 4
4 A surface area is divided into parallel strips. Elements in the strips are modelled as springs [Lefebvre and Neyret, 2002]. 5
5 Prism elements in the volumetric material layer. 6
6 Three different modes of fracture: Opening (I), in-plane shear (II) and out-of-plane shear (III). 6
7 An example triangle surface mesh. 7
8 Overview of the presented algorithm. 8
9 Decomposition of stress into tensile **(a)** and compressive **(b)** stress. 8
10 The local 2D coordinate frame (u,v) in which the stress tensor σ is defined. 9
11 The stress tensor σ in each triangle applies forces to each of its vertices. . . 9
12 Each adjacent face applies forces to a vertex. These forces can be decomposed into tensile (f^+) and compressive (f^-) parts. 10
13 Left: crack edges (red lines) are inserted at the cracking plane (red dashed line) perpendicular to the eigenvector of the crack vertex (red node). Additional edges have to be inserted to ensure a valid triangulation. Center and Right: to avoid sliver triangles, the crack edge ‘snaps’ onto an existing edge if the angle in between is under a threshold. 11
14 Cracking results for different weightings of the residual value α. Small values result in small, jagged edges which can be seen on the left, whereas large values create longer cracks which move further in the same direction, seen on the right. 13
15 An example triangle mesh before and after relaxation. Eigenvalues of the stress tensors are depicted as length of bars in the center of the triangles, the direction is determined by the corresponding eigenvector. 14
16 Left: algorithm run on a regular mesh creates a unnatural regular appearance. Right: algorithm with same parameters on irregular mesh. 15
17 The user control over the different steps of the presented method and the corresponding parameters. 16
18 When rendering crack edges directly, a gradient can be applied to the line to create a ceramic-like crack resemblance. 17
19 A visual gap is created by changing the mesh at the crack location. The width of the gap is determined by the summed up movement of each node during the relaxation process. 17

20 Sketch-based stress initialization. The left image shows an example stroke (red) and the triangles which are intersected by this stroke (yellow). Using the direction between adjacent faces as stress direction results in a jiggled direction, as seen in the center. By calculating the average direction of the n nearest neighbours in both directions on the right, the overall direction gets smoother. 19
21 A user-sketched path is used to initialize the stress field. A resulting crack pattern can be seen on the right. 19
22 Left: bark texture with a vertical crack pattern. Right: stroke sketches defined upon this texture to initialize stress at positions of existing cracks in similar directions. 20
23 The vectors used for light calculation in the Blinn-Phong model. 21
24 The influence of the exponent on the specular light calculation: small values create a large reflection and vice versa. 23
25 Normal mapping applied to a brick wall texture: the left image shows the normal map for a given brick wall texture. The brick wall texture is rendered without (center) and with (right) normal mapping. 24
26 The faces around a crack (marked red) are tessellated with the built-in OpenGL 4.0 functionality. 25
27 Crack edges divided into different connected components (color coding). . . 26
28 A crack rendered with the proposed method incorporating displacement mapping and tessellation. Triangles close to the crack edge are tessellated into smaller triangles. These are displaced depending on the distance to the crack, resulting in a V-shaped profile. 29
29 Two tetrahedral meshes are propagated into the direction of each other. When the interfaces meet, the propagation of the vertices at these positions stops, leaving the boundary intact (Image: Purdue University). 33
30 Different values for c in Equation 17. 34
31 The amount of growth of the growth function is displayed as the length of the normal vectors (green). Regions with more growth at the chest, the ears, and on top of the head of the cat have been defined by the user. The resulting evolved mesh can be seen on the right (Image: Purdue University). 36
32 Texture coordinates are propagated on an evolving mesh from one time step to the next via interpolation (Image: Purdue University). 37
33 Tree trunk (palm texture) without and with cracks. 40
34 Tree trunk without and with cracks. Birch texture top, pine bottom. . . . 41
35 Tree bark comparison of different species between photograph (left) and synthetic result (right). Species from top to bottom: palm, birch, pine. . . 42
36 Cracked bunny. 4378 triangles, 7s computation time. 43
37 Cracked mug. 12845 triangles, 33s computation time. 44
38 Cracked knot (13110 triangles, 111s computation time). Cracked gargoyle (50000 triangles, 206s computation time). 44
39 Cracked vase. 25941 triangles, 45s computation time. 45

40 Cracked neptune statue (56112 triangles, 270s computation time). Cracked mouse figure (41602 triangles, 610s computation time). 45
41 Bunny before and after mesh evolution. The growth has been defined in the direction of the surface normals. 46
42 Tree growing around fence. The DSC method stops the interface propagation when an obstacle is met. 46
43 Time-lapse style animation: knot. 47
44 Time-lapse style animation: cup. 48

List of Tables

1 Simulation Parameters: Total iterations (TI), Cracks per iteration (CPI), Relaxations per iteration (RPI), Residual value (alpha), Stress alleviation factor (SA), Random stress tensor rotation (RR), used stress pattern and computation time. 49

References

[Adalsteinsson and Sethian, 1995] Adalsteinsson, D. and Sethian, J. A. (1995). A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277.

[Barnes et al., 2009] Barnes, C., Shechtman, E., Finkelstein, A., and Goldman, D. B. (2009). Patchmatch: A randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3):24:1–24:11.

[Bernardini et al., 1999] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 5(4):349–359.

[Blinn, 1977] Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198.

[Bloomenthal, 1985] Bloomenthal, J. (1985). Modeling the mighty maple. *SIGGRAPH Comput. Graph.*, 19(3):305–311.

[Bridson, 2003] Bridson, R. E. (2003). *Computational Aspects of Dynamic Surfaces*. PhD thesis, Stanford, CA, USA. AAI3090563.

[Chopp, 1993] Chopp, D. L. (1993). Computing minimal surfaces via level set curvature flow. *J. Comput. Phys.*, 106(1):77–91.

[Cignoni et al., 1998] Cignoni, P., Montani, C., Scopigno, R., and Rocchini, C. (1998). A general method for preserving attribute values on simplified meshes. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 59–66, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Cohen et al., 1998] Cohen, J., Olano, M., and Manocha, D. (1998). Appearance-preserving simplification. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 115–122, New York, NY, USA. ACM.

[Cook, 1984] Cook, R. L. (1984). Shade trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 223–231, New York, NY, USA. ACM.

[Droske et al., 2001] Droske, M., Meyer, B., Rumpf, M., and Schaller, C. (2001). An adaptive level set method for medical image segmentation. In *Proceedings of the 17th*

International Conference on Information Processing in Medical Imaging, IPMI '01, pages 416–422, London, UK, UK. Springer-Verlag.

[Enright et al., 2002] Enright, D., Fedkiw, R., Ferziger, J., and Mitchell, I. (2002). A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116.

[Federl and Prusinkiewicz, 2002] Federl, P. and Prusinkiewicz, P. (2002). Modelling fracture formation in bi-layered materials, with applications to tree bark and drying mud.

[Finkel and Bentley, 1974] Finkel, R. and Bentley, J. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9.

[Fung, 1965] Fung, Y. (1965). *Foundations of solid mechanics*. Prentice-Hall international series in dynamics. Prentice-Hall.

[Gersho and Gray, 1991] Gersho, A. and Gray, R. M. (1991). *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA.

[Glimm et al., 1995] Glimm, J., Grove, J. W., Li, X. L., Shyue, K.-M., Zeng, Y., and Zhang, Q. (1995). Three dimensional front tracking. *SIAM J. Sci. Comp*, 19:703–727.

[Hart and Baker, 1996] Hart, J. C. and Baker, B. (1996). Implicit modeling of tree surfaces. In *In Proceedings of Implicit Surfaces '96*, pages 143–152.

[Iben, 2007] Iben, H. N. (2007). *Generating Surface Crack Patterns*. PhD thesis, EECS Department, University of California, Berkeley.

[Iben and O'Brien, 2006] Iben, H. N. and O'Brien, J. F. (2006). Generating surface crack patterns. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '06, pages 177–185, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Landsberg and Sands, 2011] Landsberg, J. and Sands, P. (2011). Chapter 4 - stand structure and dynamics. In *Physiological Ecology of Forest Production*, volume 4 of *Terrestrial Ecology*, pages 81 – 114. Elsevier.

[Lefebvre and Neyret, 2002] Lefebvre, S. and Neyret, F. (2002). Synthesizing bark. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 105–116, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching cubes:

A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169.

[Losasso et al., 2006] Losasso, F., Fedkiw, R., and Osher, S. (2006). Spatially adaptive techniques for level set methods and incompressible flow. *Computers and Fluids*, 35(10):995 – 1010.

[Losasso et al., 2004] Losasso, F., Gibou, F., and Fedkiw, R. (2004). Simulating water and smoke with an octree data structure. *ACM Trans. Graph.*, 23(3):457–462.

[Misztal and Bærentzen, 2012] Misztal, M. K. and Bærentzen, J. A. (2012). Topology-adaptive interface tracking using the deformable simplicial complex. *ACM Trans. Graph.*, 31(3):24:1–24:12.

[Nielsen, 2006] Nielsen, M. B. (2006). Denmark efficient and high resolution level set simulations- data structures, algorithms and applications.

[O'Brien and Hodgins, 1999] O'Brien, J. F. and Hodgins, J. K. (1999). Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 137–146, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Osher and Fedkiw, 2003] Osher, S. and Fedkiw, R. P. (2003). *Level set methods and dynamic implicit surfaces*. Applied mathematical science. Springer, New York, N.Y.

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317.

[Sethian, 1999] Sethian, J. A. (1999). *Level Set Methods and Fast Marching Methods*. Cambridge University Press.

[Turk, 1992] Turk, G. (1992). Re-tiling polygonal surfaces. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 55–64, New York, NY, USA. ACM.

[Wang et al., 2003] Wang, X., Wang, L., Liu, L., Hu, S., and Guo, B. (2003). Interactive modeling of tree bark. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, PG '03, pages 83–, Washington, DC, USA. IEEE Computer Society.