NTNU – Trondheim
Norwegian University of
Science and Technology

Avalanche Simulations using Fracture Mechanics on the GPU

Øivind Laupstad Boge

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

Master Thesis

Avalanche Simulations using Fracture Mechanics on the GPU

Author:
Øivind Boge

Supervisor:
Dr. Anne C. Elster

HPC
Research
Group

June 12, 2014

Problem Description

When simulating avalanches, avalanche flow, using fluid dynamics has been the main focus area in the past years. However, predicting where avalanches will occur and how large the initial mass will be, is still an open topic of research.

This project will focus on using fracture mechanics in order to predict the initial mass of an avalanche, and the snow layer modelling developed in previous work will be used to predict where in the terrain the avalanche will occur. Also since these calculations are very computationally demanding, the simulations will be parallelized for GPUs.

Assignment Given: 17. January 2014
Supervisor: Dr. Anne C. Elster

Abstract

Snow is an extremely complex material due to the structure of snow crystals and how snow behaves when it is settled within snow layers. These factors makes it hard to accurately simulate how snow layers are affected by external factors like temperature, sun radiation, and several others. And the type of snow layers, and the bonding strength between them are crucial when calculating the danger of avalanches.

In this thesis we apply fracture mechanics in order to calculate where fractures are propagating in the snow layers, and by these calculations, we can try to predict where it is high danger of avalanches. This is accomplished by using the Finite Element Method which is used to model deformation in the snow layers based on the self weight of the snow, strain and stresses is then further derived which is used to calculate the so called **Energy Release Rate**. The energy release rate is then compared to the **Critical Energy Release Rate** to determine out any fracture propagation.

The Graphical Processing Unit (GPU) is utilized to speed up the calculations due to the vast amount of data which is required to accurately simulate fracture propagation in the snow layers. And due to time limitations, optimizations has been left out. However, it was found out that even though optimization was left out, the GPU is performing about 5x faster than a parallelized CPU version.

In the simulations, data found by Christian Sigrist[21] has been used. Sigrist performed fracture testing on different kind of snow specimen within a laboratory and in the field, and he found crucial parameters for both homogeneous (stable) and heterogeneous (unstable) snow, and our simulation shows that homogeneous snow does not show any fracture propagation, and the heterogeneous snow shows a lot of uncontrolled fractures.

The results in this thesis has been obtained by different kind of visualization methods that have been implemented in this project, where we can in real-time change the visualization method and also have the possibility of pausing/resuming the simulation to obtain more detailed analysis. The types of parameters that is possible to visualize is; snow density, normal stress, shear stress, a so called energy ratio, and the lengths of the fractures.

Acknowledgement

I would like to thank Dr. Anne C. Elster which has managed to assemble to HPC-lab to the current state. This lab has given me a lot of resources, and this thesis would not have reached it current state without the lab. And I would also like to thank Anne for valuable feedback on my thesis.

Contents

List of Figures

List of Tables

Listings

List of Symbols

$\dot{\epsilon}$	The change in strain w.r.t. time
ϵ	The strain (deformation) of a material
γ_p	The energy required for plastic flow in a fracture processes
γ_s	The surface energy of a material
$\mathcal{G}$	The energy release rate in a fracture process
$\mathcal{G}_c$	The energy release rate required to extend a fracture
Π	The potential energy of a material caused by internal strain and external forces
ρ	The density of a material
σ	The stress in a material
σ_f	The stress needed for fracture propagation
$\|\|x\|\|$	The euclidean norm of x
a	The fracture length
B	The thickness of the material
E	The Young's modulus
K	The stress intensity factor, abbreviated SIF
W_s	The work required to create new surfaces within a material i.e. fractures
x^T	The transpose of vector x

Chapter 1

Introduction

Avalanches are the rapid motion of snow running downhill, and the motion of the avalanches are depening on the material properties of the snow within the avalanche, and trying to figure out models that capture their behaviour is an area of ongoing study. Models describing the flow of avalanches has achieved a lot of success, by taking advantage of the resemblance of a fluid that avalanches has while it is running down a slope[27]. In these models, researchers has used fluid dynamics in order to accurately simulate avalanches running downhill. But these models do not capture the cause and actual release of an avalanche, and the fluid dynamic models are initiated by simply "dropping" a amount of snow onto the terrain, and then simulating the avalanche as a fluid. These models for simulating the flow of avalanches can however, be useful in e.g. games and movies where they want avalanches to occur.

However, as a backcountry snowboarder myself, I think that modelling where avalanches will be triggered, and how big they could be need to be focused on. As far as i know, this is something that there does not exist any models for, which describes the release of avalanches due to the stresses in the snow structure. And since snow is an extremely heterogeneous material, accurately calculating the stress in the snow layers can be difficult. Therefore, I will look into the branch of physics which models fracture behaviour in materials called "fracture mechanics", and try to implement a model which calculates the behaviour of fractures in the snow layers, and hopefully by using this approach, we can calculate how the fractures propagate in the snow, which results in how large an avalanche will be.

In the field of fracture mechanics, the equations used for calculating the propagation of fractures within a material, is depending on some material constants. And still to this date, there are a lot of unknown material constants when it comes to snow. And since snow experiences a huge variety of behaviour and material properties, due to all external factors that changes the snow[21], accurately determining the material constants for different kind of snow is a challenge.

Also, by implementing fracture mechanic simulation within the snow layers in the HPC-Lab snow simulator, it could be used to visualize how different material constants for snow effects avalanches. The HPC-Lab snow simulator is a simulator which has been used for several different master thesis, and was first developed by Saltik[20] and is a real-time simulator for simulating snow particles and its interaction with a wind field, and when the particles collide with the terrain the snow accumulates on the ground.

In Figure 1.1 on page 2[1], we can see an avalanche which has settled, and we can

[1] www.inspirefirst.com/2012/09/18/snowboarding-scott-serfas/

Figure 1.1: Avalanche, by Scott Serfas

clearly see where the fracture has propagated at the starting zone of the avalanche, and would be the ideal result of this project.

1.1 Motivation

Fracture mechanics and FEM simulations are highly computationally demanding fields, and to accurate capture the heterogeneous complexity of snow, a fine mesh is needed for the simulation. And since the simulator is running a 3D simulation, the computational demands are even higher. This is why we have taken advantage of utilizing the powerful and parallel GPU, which has enabled us to handle even larger problem sizes and still be able to use the simulator in real-time.

Snow can also be difficult to perform laboratory experiments with, due to the fragile properties of snow, and Sigrist[21] experiences this while performing his experiments, where several snow specimen was broken while transporting the snow layers to the laboratory, and the transport process also limited the size of the snow specimen that could be brought back to the laboratory. However, if we could use the HPC-Lab snow simulator to determine how material properties of snow impact the release of avalanches, then the simulator could be used as a research platform. But this would require experimental data on avalanches, which are not present to this date.

This project will also add more focus on the avalanches release problem, and would give valuable initial conditions for any avalanche flow simulations. And a complete simulation of both the release and flow of avalanches would give great benefits to a lot of different areas like; ski resorts which has to perform a lot of snow tests in order to predict the snow stability, and avoid any fatalities, and also when building new residential areas around mountains, such that avalanche simulations could predict if avalanches could occur under different weather conditions, and if so, being able to simulate if the avalanche flow would reach any residents.

1.2 Contribution

In this project, I have extended the HPC-lab snow simulator with a more precise avalanche detection model compared to previous work implemented in my specialization project[26]. Within the time scope of this project, I did not reach as far as I wanted, due to the complex behaviour of snow and also caused by a lot of new material I had to learn in order to complete the project. e.g. Fracture mechanics and the Finite Element Method.

However, fracture mechanical calculations has been implemented, which is used to calculate fracture propagation in the snow layers in order to determine where and when avalanches would be triggered in an arbitrary terrain specified by a 2D heightmap.

A lot of different visualization methods has also been implemented such that we can visualize where avalanches are about to be triggered.

1.3 Outline

Chapter 2: covers the background material for this project, where Section 2.1 includes the basic of **Fracture Mechanics** and how to calculate how fractures propagate within a material, Section 2.2 describes snow as a material all the way from snow crystals to different kind of avalanches, Section 2.3 the basics of the Finite Element Method (FEM) used is covered. Then lastly, Section 2.4 covers the CUDA technology used for the implementation.

Chapter 3: covers previous work (Section 3.1) and related work (Section 3.2) for this project, where Section 3.1.1 describes the HPC-lab snow simulator in detail, Section 3.1.2 is used to shortly look at my specialization project[26] which lead up to this thesis, and Section 3.1.3 looks at the PhD by Christian Sigrist, titled *Measurement of Fracture Mechanical Properties of Snow and Application to Dry Snow Slab Avalanche Release*[21], and it is from this work I have obtained important material constants for snow.

Section 3.2.1 covers a project which is somewhat related to this project, but can be very useful to take advantage of in the future, namly a project called SNOWPACK[3], used for modelling snow layer properties and changes over time. And Section 3.2.2 looks into another project that uses fracture mechanics in order to predict avalanche in 2D, and another approach in 3D which uses damage mechanics[22] .

Chapter 4: Covers the implementation for simulating fracture propagation in the snow layers, and the equations used for calculating the displacement, strain, stress, and the energy release rate is shown.

Chapter 5: Is used to show the results obtained from the simulations, where the results are divided into *Simulation Results* where we look at the physical aspect of the simulations performed, and in *Performance Results* where we look into the performance on different GPUs, problem size, and CUDA kernel launch configuration.

Chapter 6: Is used to discuss the results more in detail and also some minor issues present in the simulator while using certain parameters.

Chapter 7: Includes a final conclusion of the project and further work on fully integrating the implemented fracture propagation simulation with the rest of the simulation in the HPC-lab snow simulator, and various ideas to achieve this.

Chapter 2

Background

In this chapter, we will go into detail in the necessary background material, where we first in Section 2.1, will look into fracture mechanics, where we will start at the atomic level and all the way up to structures. And we will also look into the formulas of the **Energy release rate** which is the equation used to calculate fracture propagation.

Next in Section 2.2, we will discuss snow. Starting at snowflakes to the processes involved when the snow cover changes like temperature and many other effects. We will also look into different kind of avalanches, and look at some of the complex behaviour of snow.

In Section 2.3, we will look into the Finite Element Method (FEM). Where we will look at a general overview of the method, and the steps involved, and how the method is used with an simple elastic spring example, which is related to how our FEM model is build.

Then in Section 2.4, we will look at the Compute Unified Device Architecture (CUDA) developed by Nvidia, and is supported on their GPUs.

2.1 Fracture Mechanics

Fracture mechanics is the field of mechanics where propagation of fractures in materials are studied. Alan Arnold Griffith (1921) was the first to formally describe fractures in materials based on the *global energy balance criterion*. He started his research due to the breaking of glass, where he measured that the stress needed to fracture glass was 100 MPa. However, the theoretical stress needed to break atomic bounds are approximately 10 000 MPa. He then made the assumption that micro flaws in the material increased local stresses, and then found out that when an artificial flaw of length a was created in the glass, then:

$$\sigma_f \sqrt{a} \approx C \tag{2.1}$$

The above equation was valid for extreme brittle materials like glass, and the behaviour of materials which experiences ductile behaviour at the crack tip could not be calculated with the equations formed by Griffith. The major contribution in fracture mechanics, where later in 1957 when George Rankine Irwin defined the stress located at the crack tip of the fracture. His equations includes the extra energy needed for ductile behaviour, and could therefore be used for more materials.

In fracture mechanics, there are three different types of fracture modes that can occur with different kind off applied stresses (see Figure 2.1[1]). The first mode, is a so called *Opening mode* which occurs when the material is affected by a tensile force perpendicular to the fracture plane. The second mode is called *Sliding mode*, and occurs when shear forces are acting parallel to the fracture plane, but perpendicular to the crack-tip. And the third mode is called *Tearing mode* and this fracture also occurs when shear forces are acting parallel to the fracture plane and to the crack-tip.

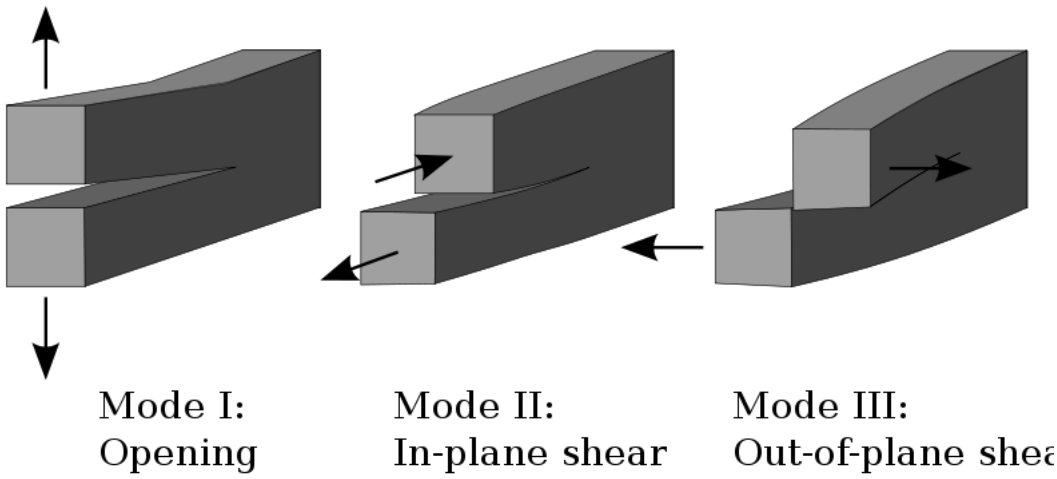


Figure 2.1: Fracture modes

In fracture mechanics, there are a lot of material properties which must be discovered in order to use this field in practice. Different materials have a huge variety in fracture properties, e.g. glass have a very brittle behaviour where there are almost no deformation of the material before any fracture occurs. On the other hand we have many metals which experiences ductile behaviour where the material is deformed before any fracture process is started. In figure 2.2[2] you can see two examples of brittle and ductile fractures, where the fractur process to the right has experiences a brittle behaviour with no deformation of the metal, and the fracture process to the left has experiences a ductile behaviour up to the point of fracture, where we can see that the metal has become thinner towards the fracture plane.

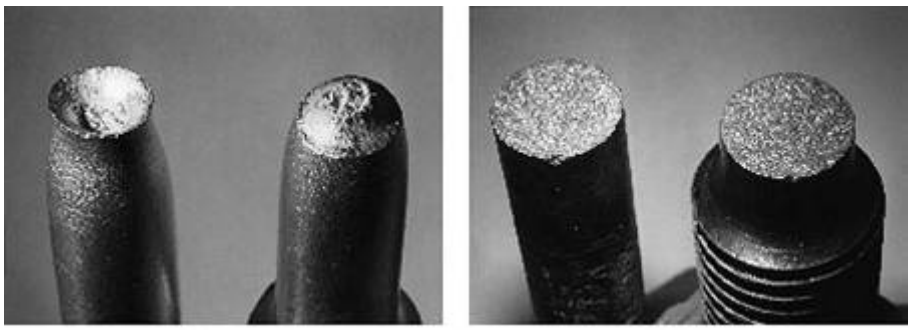


Figure 2.2: Ductile vs brittle fracture

[1]Public domain license, reprinted from Wikipedia: http://en.wikipedia.org/wiki/File:Fracture_modes_v2.svg

[2]Reproduced with permission from NSW HSC Online http://hsc.csu.edu.au © NSW Department of Education and Communities, and Charles Sturt University, 2014

Despite the substantial progress that has been made in the past decades, the theory of fracture mechanics is by far not completed[21], and since the field depends on a lot of different material constants and functions, there are still to this date a lot of unknown variables. However, several laboratory experiments are being performed in order to determine these variables, like Christian Sigrist [21] which performed a lot of experiments on different kind of snow types, and have acquired a lot of data in his PhD thesis.

2.1.1 Atomic View of Fracture

A fracture can be thought as the stress needed to break atomic bonds between atoms in a material [23]. Two atoms will have an equilibrium spacing between them of distance x_0, then to fully separate the two atoms, a tensile force is required to increase the distance between the two atoms, and this force must exceed the cohesive force between the atoms. The energy needed to separate two atoms is given by the following equation:

$$E = \int_{x_0}^{\infty} P dx$$

where x_0 is the equilibrium spacing and P is the applied force

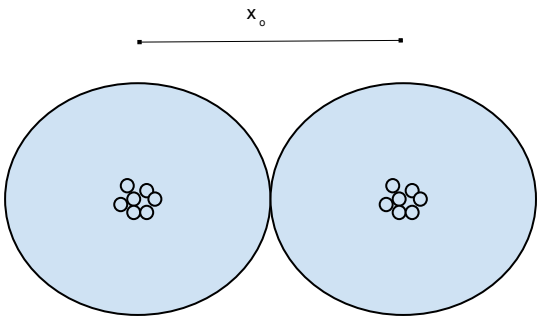


Figure 2.3: Equilibrium spacing between two atoms

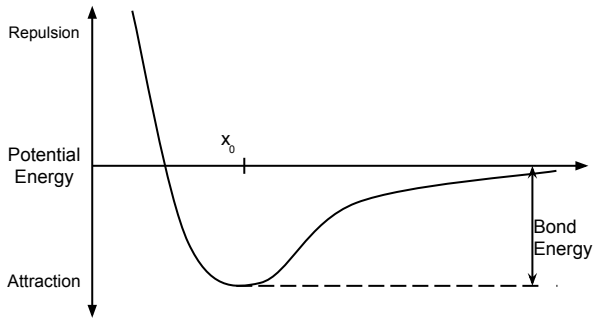


Figure 2.4: Potential Energy

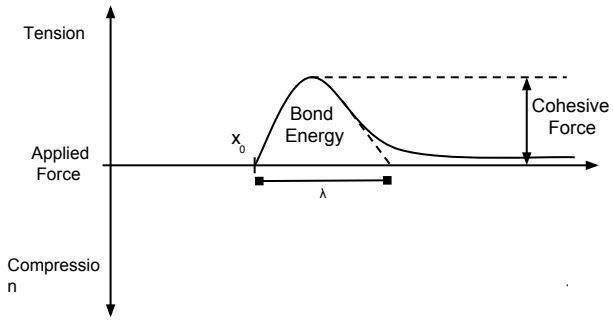


Figure 2.5: Bond Energy

It is then possible to estimate the cohesive strength between two atoms as the following:

$$P = P_c \, sin\left(\frac{\pi \, x}{\lambda}\right)$$

where λ is defined in Figure 2.5

Further estimations gives that for small displacements the force requirement is linear if we place the origin at x_0, so:

$$P = P_c \left(\frac{\pi \, x}{\lambda}\right)$$

We can then follow Hooke's law, and find the bond stiffness constant:

$$k = P_c \left(\frac{\pi}{\lambda}\right)$$

It can then be shown that the cohesive stress of a material can be calculated as the following:

$$\sigma_c = \frac{E \, \lambda}{\pi \, x_0} \approx \frac{E}{\pi} \tag{2.2}$$

E is Young's modulus

We can also calculate the cohesive stress by using the surface energy of the material. First we estimate the surface of the material by the following:

$$\gamma_s = \frac{1}{2} \int_0^{\lambda} \sigma_c \, sin\left(\frac{\pi \, x}{\lambda}\right) dx = \sigma_c \, \frac{\lambda}{\pi} \tag{2.3}$$

Then combining Equation 2.2 and 2.3 gives:

$$\sigma_c = \sqrt{\frac{E \, \gamma_s}{x_0}} \tag{2.4}$$

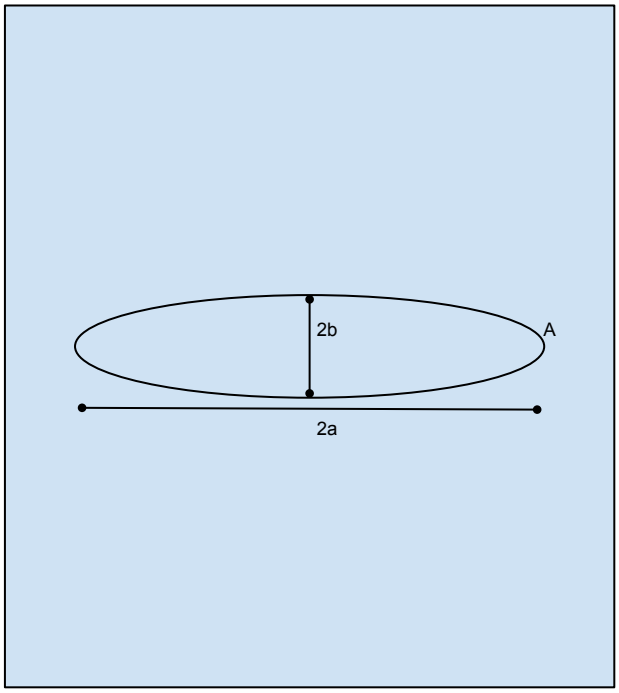


Figure 2.6: Elliptic flaw in a material

2.1.2 Effect of Microflaws

When fracture experiments was performed, they found out that the fracture strength of brittle material was approximately 3-4 times lower than the theoretical cohesive strength of the material $\sigma_c = E/\pi$. It was then assumed that this was due to flaws in the material which increased the stress locally, making the global strength of the material lower than it originally was.

The first evidence of these micro flaws within the material was performed by Inglis in 1913 [7]. He analysed elliptic holes in a flat plate illustrated in Figure 2.6. He calculated the stress concentration at a point A, and assumed that the stresses was not affected by any boundary conditions, i.e. the width of the plate $\gg 2a$, and the height of the plate $\gg 2b$. He then found an expression for the stress concentration at point A to be:

$$\sigma_A = \sigma \left(1 + 2\sqrt{\frac{a}{\mu}}\right) \tag{2.5}$$

Where $\mu = \frac{b^2}{a}$ is the radius of the elliptic curvature

However, a fracture does not have the form of a ellipse. But if we take the ellipse and make a $\gg$ b, then we will have an approximation to a sharp crack, and Equation 2.5 can be approximated to:

$$\sigma_A = 2\sigma\sqrt{\frac{a}{\mu}} \tag{2.6}$$

But from a mathematically point of view, Equation 2.6 leads to the following problem. When a is much larger than b, μ will tend to zero, and the stresses at the crack tip will be infinitely large, and no material can withstand infinitely stress. However, from a physical point of view, a fracture tip cannot have a radius

smaller than the equilibrium distance between two atoms. So the largest possible stress concentration of an atomically sharp edge will be:

$$\sigma_A = 2\sigma\sqrt{\frac{a}{x_o}} \tag{2.7}$$

Finally it is assumed that a fracture can propagate when the stress at the crack-tip is equal to the cohesive strength between two atoms $\sigma_A = \sigma_c$, so combining Equation 2.4 and 2.7 results in the following expression for the stress present at fracture propagation.

$$\sigma_f = \sqrt{\frac{E\gamma_s}{4a}} \tag{2.8}$$

Crack-tip Stresses: This is another method of analysis where a local coordinate system is defined around the crack-tip, and we can then calculate the stress field around the crack-tip in order to find out how the fracture will propagate in the material. The local coordinate system is displayed in Figure 2.7, however, this approach is only valid in a isotropic linear elastic material[3].

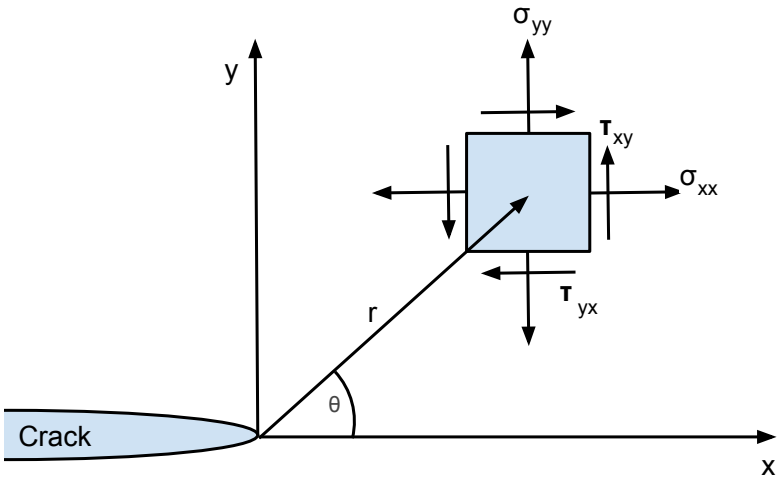


Figure 2.7: Local coordinate system at crack-tip, where σ is normal stress and τ is shear stress

It can then be shown that the stress field around the crack-tip is any linear elastic cracked body is given by[23]:

$$\sigma_{ij} = \left(\frac{k}{\sqrt{r}}\right) f_{ij}(\theta) + \sum_{m=0}^{\infty} A_m r^{\frac{m}{2}} g_{ij}{}^{(m)}(\theta) \tag{2.9}$$

[3]Isotropic materials are characterized by properties which are independent of direction in space

where:

- σ_{ij} is a stress tensor
- r and θ is defined in Figure 2.7
- k is a constant
- f_{ij} is a dimensionless function of θ

It it also very common to substitute $K = k\sqrt{2\pi}$, where K is the stress intensity factor (SIF). The stress intensity factor is a measurement of the fracture toughness, and is divided into K_I, K_{II}, and K_{III} for the different loading modes of fractures, given in Figure 2.1 on page 6. The SIF is therefore a material constant and will have to be determined for all different modes for the desired material.

We can then use Equation 2.8 and 2.9 on page 10 for calculating fracture propagation. When the stress field around the crack-tip is equal to the σ_f, the fracture can grow a distance r in θ direction.

2.1.3 Energy Balance

In 1920, A. A. Griffith used the first law of thermodynamics to describe the formation of fractures [4]. This law states that a system going from a nonequilibrium state to equilibrium will always have a net decrease in energy. Griffith used this approach because previous estimation of fracture propagation described in Equation 2.8 is only a rough estimate of the failure stress, because the equation is only valid for ideally brittle materials[23]. Griffith expressed the propagation of fractures in the following way:

$$\frac{dE}{dA} = \frac{d\Pi}{dA} + \frac{dW_s}{dA} = 0 \Rightarrow -\frac{d\Pi}{dA} = \frac{dW_s}{dA} \tag{2.10}$$

where E is total energy, Π is potential energy supplied by internal strain and external forces, and W_s is work required to create new surfaces i.e. fractures, and A is the area of the fracture surface

He then used the analysis of Inglis [7] to show that:

$$\Pi = \Pi_0 - \frac{\pi\sigma^2 a^2 B}{E} \tag{2.11}$$

and

$$W_s = 4aB\gamma_s \tag{2.12}$$

where Π_0 is the potential energy of an uncracked plate, and B is the thickness of the plate. If we then insert this into Equation 2.10 we get the following expression for the stress present at failure:

$$\sigma_f = \sqrt{\frac{2E\gamma_s}{\pi a}} \tag{2.13}$$

The approach used by Griffith is fairly consistent with the local stress criterion by Inglis, and they do not differ more than 40% [23]. But they both require a sharp fracture in a brittle material.

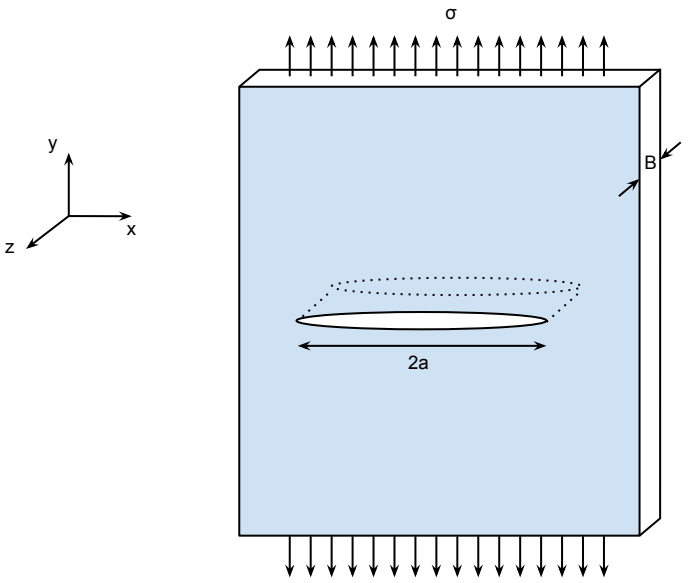


Figure 2.8: Micro-flaw in a material

Irwin's modification: A modification of griffith equations was later performed in 1948 due to the severe underestimation of fracturing of metals by G. R. Irwin[8], because Equation 2.13 on page 11 is only valid for ideally brittle materials. Irwin then formulated an equation which takes plastic flow into account, and is expressed as following:

$$\sigma_f = \sqrt{\frac{2Ew_f}{\pi a}} = \sqrt{\frac{2E\left(\gamma_s + \gamma_p\right)}{\pi a}} \tag{2.14}$$

Where γ_p is the energy wasted on plastic flow, and γ_s is the surface energy, and we can see that the equation is essentially equal to Equation 2.13.

The Energy Release Rate: The work performed by Griffith was mostly ignored by other engineers because they required extreme brittle materials like glass, and the equations ignore any plastic deformation at the crack-tip, which can even appear in materials which seems brittle.

In 1956, Irwin proposed an *'Energy Release Rate'* [9], which is equivalent to Griffiths approach but is more convenient to use when solving engineering problems. The energy release rate proposed is a measure of the energy available for an increment of fractures, and is defined as the following:

$$\mathcal{G} = -\frac{d\Pi}{dA} = -\frac{d\left(\Pi_0 - \frac{\pi\sigma^2 a^2 B}{E}\right)}{dA} = \frac{\pi\sigma^2 a}{E} \tag{2.15}$$

for a wide plate in plane stress with a crack of length $2a$ (Figure 2.8)

It can also be shown that Equation 2.15 holds for both load controlled fractures, and displacement controlled fractures[23]. And when $\mathcal{G}$ reaches a critical value $\mathcal{G}_c$ the fracture can propagate a given distance. And $\mathcal{G}_c$ is defined by:

$$\mathcal{G}_c = \frac{dW_s}{dA} = 2w_f \tag{2.16}$$

Where w_f can either be equal to the surface energy of a brittle material $w_f = \gamma_s$, or $w_f = \gamma_s + \gamma_p$ for materials experiencing plastic flow at the crack-tip.

2.1.4 Global $\mathcal{G}$ and local SIF

The global energy release rate $\mathcal{G}$ and the local crack-tip stress analysis, uses two quite different approaches for solving fracture propagation. The global energy release rate calculates the change in potential energy w.r.t. a change in fracture area, and the potential energy is calculated by the stress in the structure, generated by external forces and the size of the fracture. And the fracture will then propagate when the energy release rate is equal to the critical energy release rate $\mathcal{G}_c$, which is the work required to generate new surfaces in the material.

Secondly we have the local approach which defines a coordinate system around the crack-tip. This method calculates the stress field around the crack-tip to solve the fracture propagation, and uses the stress intensity factor to characterize the stresses at the crack-tip. And it can be shown that the local K constant and the global $\mathcal{G}$ has a unique relationship defined as the following:

$$\mathcal{G} = \frac{K^2}{E} \tag{2.17}$$

This mean that we can determine the critical energy release rate $\mathcal{G}_c$ from the critical stress intensity factor K, and this factor has been determined by Sigrist[21] for various types of snow, which we will discuss more in Section 3.1.3.

2.2 Snow

Snow is the natural phenomena occurring when small amounts of water is supercooled in the clouds and freezes into small hexagonal prisms, and as they fall towards the earth they will continue to grow due to water vapor in the air[12]. As the prism falls towards the earth and continuously gets in contact with water vapor, it will gradually grow into more and more complex structures, as the surrounding water vapor condenses directly into ice on the surface on the prism. The factors which contribute to the amount of the complex growth, and the form of the snow crystals have been researched by Kenneth G. Libbrecht at Caltech[12], and he concluded that the main factors that differs the geometry and growth of snow crystals are mainly air temperature and humidity, and his findings are shown in Figure 2.9 on page 14 [4].

After a winter season of snowing, the result can be several snow layers with a lot of different mechanical properties, due to weather conditions throughout the winter season. A snow layer is formed after a given amount of snow has settled on the ground with given properties due to the current weather condition. And when it starts snowing again, a new snow layer will form with a different thickness due to the amount of snow, and also different properties due to weather conditions.

There are also other factors which contribute to the properties of the snow layers, like; radiation from the sun acting on the surface of the topmost snow layer, wind transportation of the light snow crystals on the surface of the snow, and sintering in the snow layers. All these factors also result in a very heterogeneous snow structure, and as Christian Sigrist has measured [21], the fracture toughness

[4] www.snowcrystals.com

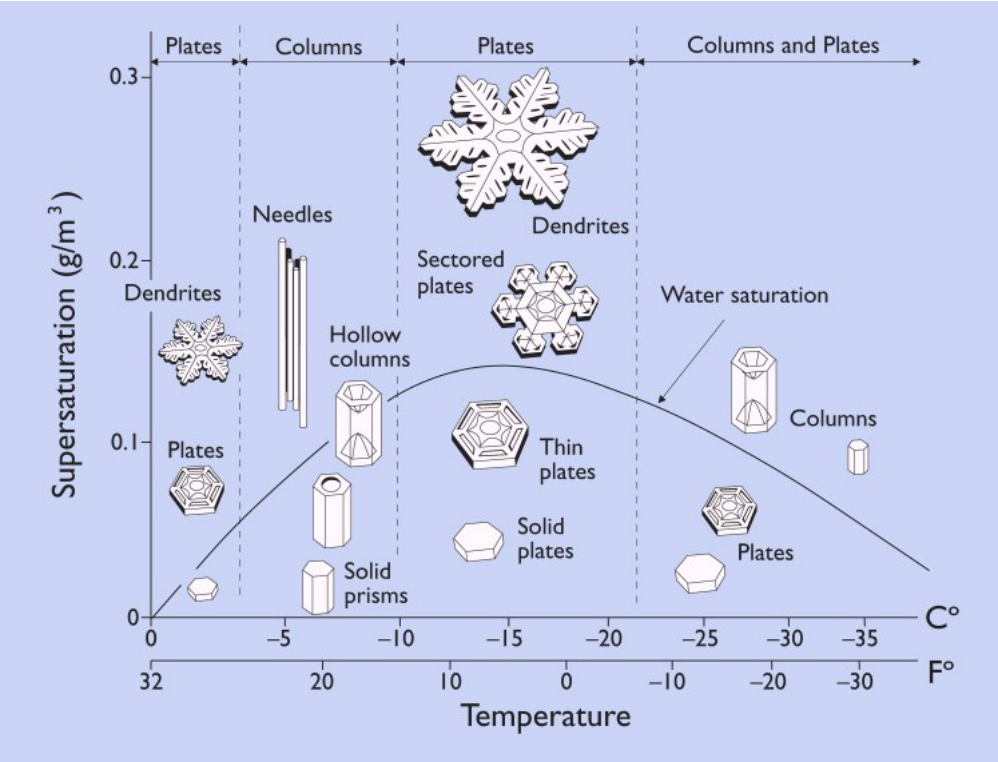


Figure 2.9: Morphology diagram for snow crystals, by Kenneth G. Libbrecht

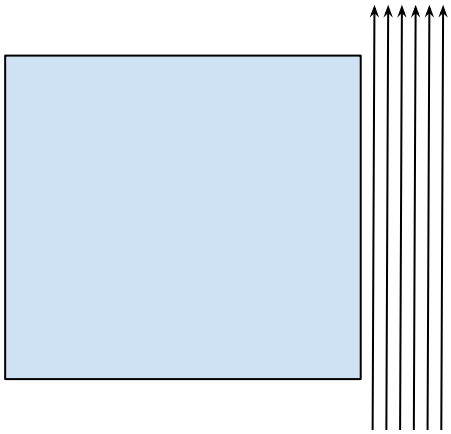
in a heterogeneous snow structure is not that good, compared to a homogeneous snow structure. Where a homogeneous snow structure is a single snow layer with given machanical properties, and a heterogeneous snow structure is the interface between snow layers. Which means that fractures can more easily propagate in the interface between snow layers.

2.2.1 Material Properties

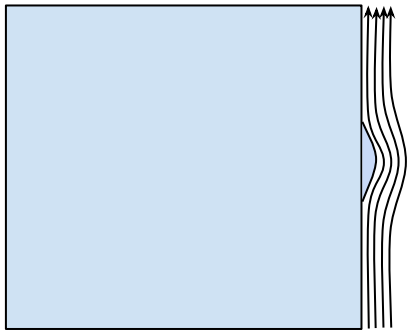
As mentioned before, snow is a very complex material and has a lot of factors determining its properties. We will now look into some of the factors.

Temperature: Temperature has a great impact on snow, and when the temperature in the snow increases it has two effects[21]. Firstly it has a long term effect which causes the stability to increase due to the bonds between snow crystals being able to grow more quickly, secondly it also has a short term effect causing the snow layers to decrease in stiffness, which favors the propagation of fractures.

Sintering: The process called sintering, has a big impact on snow and how we experience snow. Sintering is a thermal process where particles are bonded together through mass transportation events[21]. A more practical example of this effect is, when two cubes of snow are placed besides each other, they will automatically within seconds begin to make bonds to each other without any external applied force. And snow may be the only material which experiences this naturally, because snow exists so closely to its melting point. But from a fracture point of view, this sintering process creates the possibility for snow to *'heal'* fractures, as long as the fracture surfaces are in contact with each other.



(a) Wind field at starting point



(b) Wind field with bump at one side of the hexagonal prism

Figure 2.10: Snow crystal branching

Complex structure: Individual snow crystals has an extremely complex structure, which was discussed some in Section 2.2, but now we will look more into how the structure of snow crystals are formed. When snow crystals are formed, they start out as hexagonal prisms and then gradually grow more and more complex as the surrounding water vapor condenses into ice, and this process have a natural way of adding more complexity to existing complexity. Consider the starting point of an snow crystal displayed in Figure 2.10a, where the water vapor will flow in parallel to the sides. Then after a while, some of the water vapor will condense into ice and attach itself to one of the sides. Then after this process, the surrounding water vapor will now travel along a different and longer path displayed in Figure 2.10b.

This results that the original time the water vapor used to travel over the area without a bump will increase as the bump increases, and more and more water vapor will condense into ice in this area, which results in complex branching in this area.

Brittle and ductile: Snow will act like a brittle material or ductile material depending on the strain rate. Which is the rate of change in strain (deformation) in a material, and is calculated by the following:

$$\epsilon = \frac{L - L_0}{L_0}$$

Where L is the deformed length of the material, and L_0 is the original lenght of the material, and ϵ in strain

$$\dot{\epsilon} = \frac{d\epsilon}{dt}$$

Where $\dot{\epsilon}$ is the strain rate

If we deform snow with $\dot{\epsilon} > 10^{-3}s^{-1}$, snow will act like a brittle material. But on the other side, if we use a lower strain rate of $\dot{\epsilon} < 10^{-5}s^{-1}$, snow will act like a ductile material[21]. However, the process of an avalanche has very high strain rate[21], and therefore we should be able to consider snow as a brittle material.

Specific strength: The specific strength of snow is extremely low. The specific strength is defined as the forces per unit area of failure, divided by the density of the material:

$$S = \frac{\frac{F}{A}}{\rho} = \frac{\sigma_f}{\rho}$$

Jamieson and Johnston (1990)[21] did some research on the specific strenght of snow, and found values ranging between $2Nm/kg$ and $23Nm/kg$. These are extreme low values when compared to other materials like ice, which has a specific strenght of $10000Nm/kg$, or aluminium which has $30000Nm/kg$.

Metamorphism: Metamorphism is highly complex when it comes to snow. Parts of the snow will sublimate[5] into water vapor, and start to rise upwards in the snow layers and will condensate at a different location. This results that snow are always changing, and snow grains will change as time passes. It is also found that different snow grain types can have the same density [21], while it has different mechanical properties, therefore the density alone is not sufficient to describe the mechanical properties of snow.

2.2.2 Avalanche

Avalanches are the phenomena where large amount of snow starts to slide down a mountain side, and can either be triggered naturally or by skiers. Figure 2.11[6] on page 17 displays a avalanche running downhill with a powder cloud on top of it. And as we will discuss, there are several different types of avalanches, but there are no global classification, however some are commonly used.

Classification: The classification of avalanches are typically made into two different classes which are *slab avalanche* and *loose snow avalanche*. There are others as well, however these two are the main classes which has their focus on the actual release of the avalanche. There are for example another class of avalanche called *powder snow avalanche*, which has the characteristics of a powder cloud running on top of the avalanche as displayed in Figure 2.11, and another one called *wet snow avalanche*, which has the characteristics of a fluid running down the slope, and the snow is very heavy and contains a high concentration of water. However, the two latter classifications does not consider the release itself, they focus on the flow characteristics of the avalanche. And therefore we will only describe the slab avalanches and loose snow avalanches in this thesis.

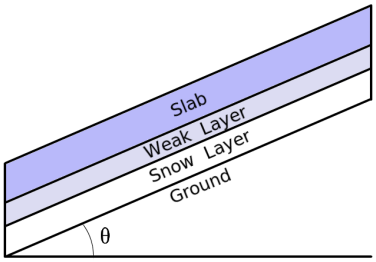
[5]Transition of a material from solid directly into gas

[6]Creative Commons Attribution-Share Alike 3.0 Unported, reprinted from Wikipedia http://en.wikipedia.org/wiki/File:2007-02-15-CLB-Couloir2-1c.JPG

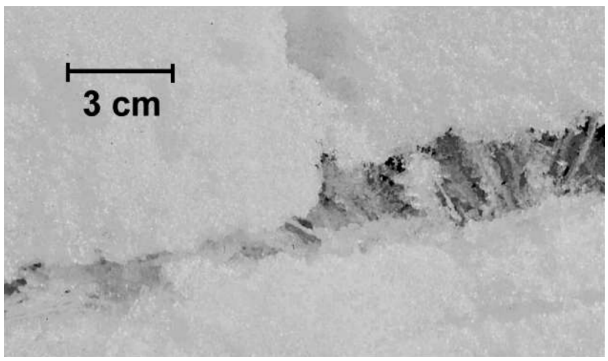
Figure 2.11: Avalanche

Loose snow avalanches: This class of avalanches are the type of avalanche where relative small amount of snow starts to slide down the terrain. These types of avalanches are most rapidly when terrain slopes is above 40 degree, persistent sub-zero temperature, and low humidity[25] (see Figure 2.12). However, this class of avalanches do not give a clear view of the fractures, and using fracture mechanics to predict these avalanches are probably not the best approach. But loose snow avalanches are not the most dangerous class of avalanches to backcountry skiers. Experienced skiers have to deal with these avalanches all the time when skiing in very steep slopes. And this class of avalanche is not that fatal compared to slab avalanches, but there have been reports of loose snow avalanches which have steered skiers off clips.

Figure 2.12: Loose snow avalanche, with permission from Canadian avalanche centre, photo by: Jim Bay



(a) Weak layer triggering slab avalanches



(b) Weak layer photo, photo by: ASARC (Applied Snow and Avalanche Research) University of Calgary, Canada

Figure 2.13: Weak layer, main cause of slab avalanches

Slab avalanches: Slab avalanches are the most dangerous type of avalanches. These types of avalanches form when several different kind of snow layers are formed, as the result of a winter season with snow falls consisting of different weather conditions. And in the interface between snow layers, there can exist a *weak layer*, where fractures can more easily propagate. This theory of *'weak layers'* is globally accepted, and was tested in[22], and states that fractures can more easily propagate in these weak snow layers, resulting in huge amount of snow masses that suddenly starts to slide downhill. But this theory is yet to be proven. In Figure 2.13a is a schematically drawing of a weak layer with an overlying slab layer, and in Figure 2.13b you can see an actual photo of an weak layer, and we can see that these weak layer are extremely porous and probably extremely fragile. In Figure 2.14 [7] the size of a slab avalanches is shown, and in this figure we can also clearly see the fracture which seperates the avalanche from the rest of the snow which did not start to slide downhill.

Figure 2.14: Weak layer triggering slab avalanches

[7]Public domain image, reprinted from National Park Service - U.S Department of the interior http://www.nature.nps.gov/geology/hazards/avalanche.htm

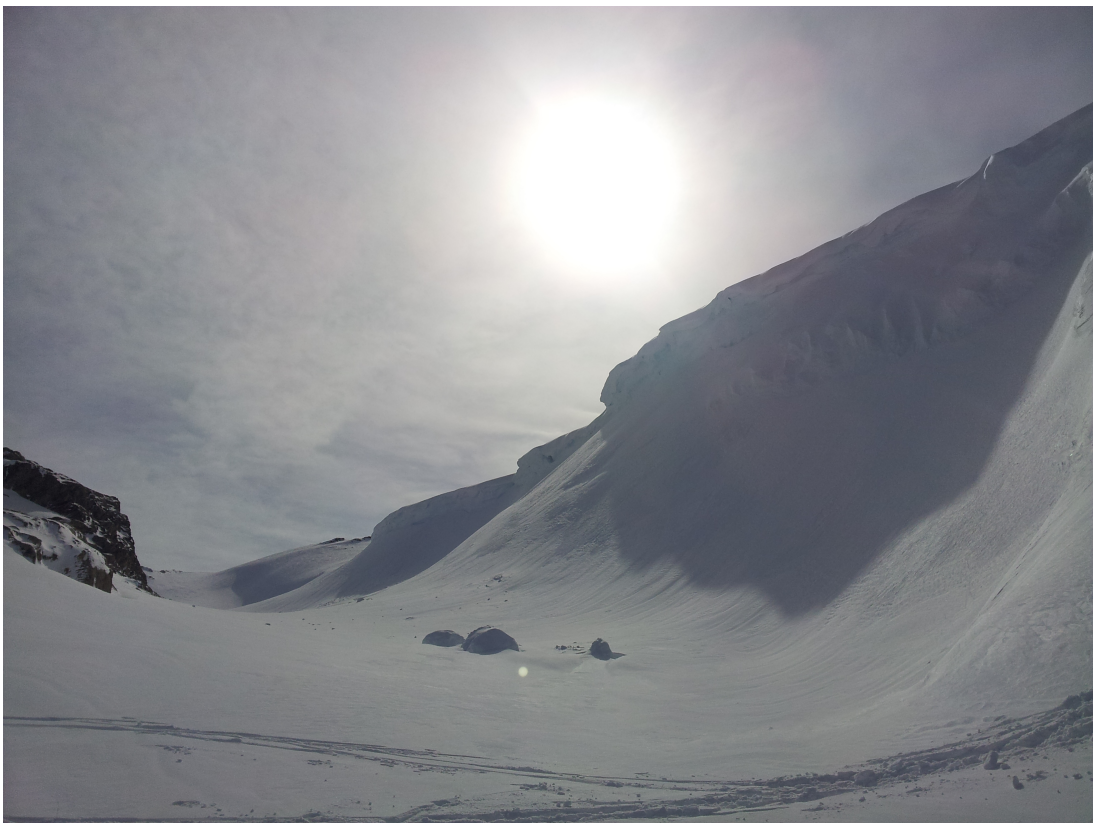


Figure 2.15: Snowdrift, Photo taken by the author at Folgefonna, Norway, 2012

2.2.3 Behaviour of Snow

In Section 2.2.1, we looked into snow as a material and which parameters that contribute to the material properties. We will now look at some examples of the behaviour of snow, and how complex it can be.

When looking over a large time span of an entire winter, or several winters, snow can change into extreme dangerous structures that are extremely fragile. In Figure 2.15 it is shown a terrain where the snow cover in the upper right corner has formed a *'snowdrift'*, and if the weight of a human is added on top of this structure, it would most likely collapse, and possible initiate an avalanche dependent on the surrounding terrain.

Snow also have a very good probability when it comes to sticking itself on the surfaces on other materials, as displayed in Figure 2.16 on page 20, where we can see that the trees are almost 100% covered in snow. When snow manages to stick large amount of snow on other surfaces, it is difficult for the first layer to attach itself. However, after the first layer has been attached, subsequent snow layers can more easily stick to the existing snow. And if a large amount of snow is attached, the mass of the snow will increase, and could eventually fail. And this could be dangerous if the snow has attached itself outside a large cliff, and then suddenly a huge amount of snow is dropped down the cliff.

(a)

(b)

Figure 2.16: Trees covered in light snow, Photos taken by the author in Nelson, British Columbia, Canada, 2013

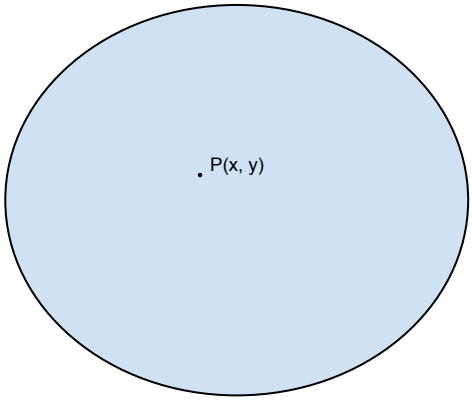


Figure 2.17: FEM domain

2.3 Finite Element Method

The Finite Element Method (FEM) is a computational technique for obtaining approximate solutions of a boundary value problem, which is a mathematical problem where one or more dependent variables must satisfy a known differential equation everywhere in the specified domain of independent variables, and also satisfy the conditions at the boundary [6].

When solving these problems, the values of the dependent variables are specified on the boundary also known as the *boundary condition*, and the complete system can then be solved because all internal points in the domain must satisfy a differential equation. These problems are also called *field* problems, where you have a *field variable* which is the dependent variable governed by the differential equation, and the boundary condition is then the specified field variables at the boundary of the domain, and depending on the type of problem being analysed the field variable could be physical displacement, temperature, heat flux, or fluid velocity.

2.3.1 Introduction

As stated before, FEM tries to approximate the solution of a field variable across a domain. Imagine that we want to find the solution of a 2D function $\phi(x, y)$ at every point $P(x, y)$ over the domain in Figure 2.17, and a known differential equation is satisfied at every point.

The finite element method is then used to create a number of finite elements across the entire domain as shown in Figure 2.18 on page 22. Each element is then represented by a number of nodes, where the value of the field variable is calculated. Nodes can either be exterior nodes, which are nodes located on the boundary of the finite elements, and exterior nodes are therefore used to connect several finite elements. Or nodes can be interior nodes which is located within a finite element and cannot be used to connect other finite elements. The element in Figure 2.18 has 3 exterior nodes.

After the field variable has been calculated for all nodal points (nodes), these values are used to approximate the value at any point within each element. And we can then use interpolation to find the approximation everywhere within each finite element:

$$\phi(x, y) = N_1(x, y)\phi_1 + N_2(x, y)\phi_2 + N_3(x, y)\phi_3 \tag{2.18}$$

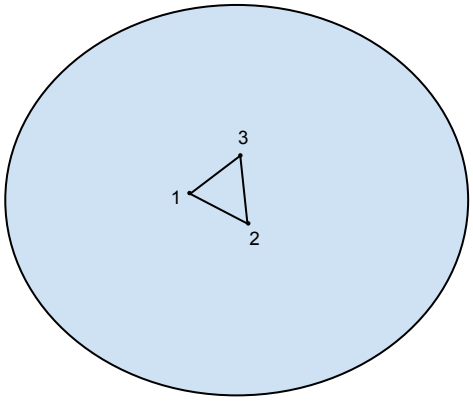


Figure 2.18: Finite element within domain

Where ϕ_1, ϕ_2 and ϕ_3 are the values of the field variable at the nodes, and N_1, N_2 and N_3 are the interpolation functions, and these interpolation functions are predefined functions of the independent variables, and describes the variation of the field variable across a single finite element.

The triangular element which we have looked at so far is also said to have 3 degrees of freedom, since it requires the calculation of 3 nodal points in order to determine the field variable within the entire finite element as displayed in Equation 2.18 on page 21. But this would only be the case if the field variable was a scalar value, like temperature. If the domain where representing a thin solid body subjected to plane stress, the field variable would be a 2D vector representing displacement within the body, and therefore we would have 6 degrees of freedom.

However, it is not always the actual field variable that we are interesting in when using the finite element method, e.g. if we want to calculate the strain and stresses within a body we must first solve the displacement field variable, and then solve for strain, which is the derivative of displacement w.r.t. time, and then solve the stresses within the body by the strain-stress relationship defined by Young's modulus. These variables are also known as *derived variables*, however these are not necessarily continuous within the domain of analysis, but the primary field variable (in this case displacement) is always continuous within the domain.

2.3.2 General Overview

When using finite element method to solve various problem like heat transfer or structure analysis, the general approach for implementing the method is always the same, and in this section we will look at the general overview of the finite element method.

Preprocessing: The first step of any FEM analysis is preprocessing, and this is the step where we define the following:

- The geometric domain of the problem
- The type(s) of elements that are being used
- The material properties of the elements
- The geometric properties of the elements, like lengths, area, or volume

- The element connections, i.e. exterior nodes
- The physical constraints
- The loadings

Solution: During this phase, the primary field variable is solved across the domain. This is accomplished by defining the governing equations in matrix form and solving for the unknown field variable. These values are then used by back substitution to compute additional derived variables.

Postprocessing: This is the final step of the finite element method, and consist of analysis and evaluation of the results.

2.3.3 The Stiffness Matrix

In Section 2.3.2, we looked briefly at the overview of the finite element method, we will now go more into details of the preprocessing step of the method, where we will look at the *stiffness matrix*.

If we apply the finite element method to a structural problem, where the main goal is the model deformation of a structure caused by applied forces, then the stiffness matrix is relative easy to understand. In this case, the stiffness matrix represents the elements resistance to deformation such as axial deformation[8], bending[9], shear deformation[10], and torsional effects[11]. But the stiffness matrix term is also used when solving all kinds of problems with the finite element method, and in other cases like heat transfer the representation of the stiffness matrix may not be that clear. However, the general representation of the stiffness matrix is the elements resistance to change when subjected to external influences.

Figure 2.19: Spring system consisting of 1 one-dimensional element

We will now look at a concrete example of the stiffness matrix [6]. In Figure 2.19 the finite element mesh of a 1D linear elastic spring is modelled. The entire domain consists of only 1 finite element, where the finite element is represented by two exterior nodes, and each node has an applied force f and a displacement u. A linear elastic spring also have the property that when a deformation of length Δu is applied, the *net force* stored in the spring is given by the following equation:

$$f = k\Delta u \tag{2.19}$$

[8]Deformation with applied force perpendicular to given axis

[9]Deformation with an external load applied perpendicularly to a longitudinal axis of the element

[10]Deformation due to shear stresses

[11]Twisting of an object due to an applied torque

where k is the spring constant

The resulting net force in the spring when a given two nodal displacements u_1 and u_2 is then:

$$f = k\Delta u = k\left(u_2 - u_1\right) \tag{2.20}$$

Further we know that for equilibrium conditions $f_1 + f_2 = 0 \rightarrow f_1 = -f_2$, and we can then rewrite Equation 2.20 to the following set of equations:

$$f_1 = -k\left(u_2 - u_1\right)$$

$$f_2 = k\left(u_2 - u_1\right)$$

Which again can be expressed in matrix format:

$$\begin{bmatrix} k & -k \\ -k & k \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{2.21}$$

the above K-Matrix will in this case be the stiffness matrix, and is used to calculate the displacement with applied forces. The stiffness matrix in this case also is a 2×2 matrix, this is because that each element (only one in this case) experiences 2 nodal displacements (degrees of freedom), and that they are dependent on each other. We can also see that the matrix is symmetric, and this is a general results of the finite element method. A finite element consisting of N degrees of freedom, will have a $N \times N$ symmetric stiffness matrix.

2.3.4 Global Assembly

In the previous Section 2.3.3, we defined the element stiffness matrix for a system consisting of 1 element in equilibrium state, the same approach can also be used for a system consisting of several elements. We will now consider the system displayed in Figure 2.20, where we have an elastic spring consisting of 2 finite elements, and 3 nodes N_1, N_2 and N_3, and the two elements are connected at node N_2. These nodes will have a global behaviour, which are the global displacement U_1, U_2, and U_3, and the global forces are F_1, F_2, and F_3. The finite elements will also still have their local forces and displacements $f_k^{(i)}$ and $u_k^{(i)}$, where k is the index of the property, and (i) is the element number. Note that uppercase is used for global behaviour, and lowercase is used for local element behaviour. We can then do the same approach as in Section 2.3.3 for each element, and describe the local

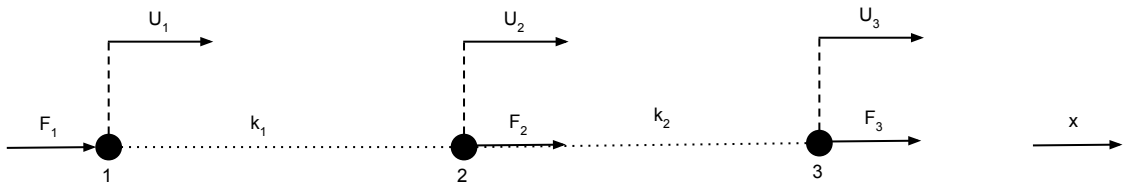


Figure 2.20: Spring system consisting of two spring elements

displacement resulting in a local force:

$$\begin{bmatrix} k_1 & -k_1 \\ -k_1 & k_1 \end{bmatrix} \begin{Bmatrix} u_1^{(1)} \\ u_2^{(1)} \end{Bmatrix} = \begin{Bmatrix} f_1^{(1)} \\ f_2^{(1)} \end{Bmatrix} \tag{2.22a}$$

$$\begin{bmatrix} k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} u_1^{(2)} \\ u_2^{(2)} \end{Bmatrix} = \begin{Bmatrix} f_1^{(2)} \\ f_2^{(2)} \end{Bmatrix} \tag{2.22b}$$

To start the global assembly, we must first relate the local displacements $u_k^{(i)}$ at each element by the global displace U_j. The relation between these variables are as follow:

$$u_1^{(1)} = U_1 \qquad u_2^{(1)} = u_1^{(2)} = U_2 \qquad u_2^{(2)} = U_3$$

Which gives us the following expressions for each element:

$$\begin{bmatrix} k_1 & -k_1 \\ -k_1 & k_1 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} f_1^{(1)} \\ f_2^{(1)} \end{Bmatrix} \tag{2.23a}$$

$$\begin{bmatrix} k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} f_1^{(2)} \\ f_2^{(2)} \end{Bmatrix} \tag{2.23b}$$

In Equation 2.23, we can clearly see that both of the finite element depend on the exact same displacement U_2, which is the node that connects the two element. The next step in the global matrix assembly is then simply expanding the matrices in Equation 2.23:

$$\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ 0 \end{Bmatrix} = \begin{Bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ 0 \end{Bmatrix} \tag{2.24a}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & k_2 & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} 0 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ f_1^{(2)} \\ f_2^{(2)} \end{Bmatrix} \tag{2.24b}$$

Next, we add the two matrix systems together to form the systems of equations for the entire structure.

$$\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 + k_2 & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_2^{(2)} \end{Bmatrix} \tag{2.25}$$

Then finally we need to express the matrix in Equation 2.25 by the global force F_1, F_2, and F_3, where we have the following relationship between the local force $f_k^{(i)}$ and F_i:

$$F_1 = f_1^{(1)} \qquad F_2 = f_2^{(1)} + f_1^{(2)} \qquad F_3 = f_2^{(2)}$$

Which gives us our final global system of equations:

$$\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 + k_2 & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \end{Bmatrix} \tag{2.26}$$

The K matrix in Equation 2.26 is now the system stiffness matrix, which represents the entire structure consisting of two finite elements connected at a single node. Also note that this is a symmetric matrix, and is simply a superposition of the element stiffness matrices.

2.4 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a development toolkit developed by Nvidia and was first released in 2007. This development toolkit includes a *Nvidia Cuda Compiler (NVCC)*, which is used for compiling a relative high-level language called CUDA C/C++ into PTX code, which is a low-level parallel thread execution virtual machine and instruction set architecture (ISA) supported by Nvidia GPUs. The toolkit is also shipped with a profiling tool, and a set of highly optimized libraries like:

- cuFFT
- cuBLAS
- cuSPARSE
- and several more

The CUDA toolkit has been released for several different versions, and is currently at version 6.0 which was released April 2014, where the latest news is the *Unified Memory* which makes it easier for a programmer to access the same memory from both the CPU and GPU and does not require to explicitly manage memory transfer between them. However, this feature is not used in this project, and therefore we will not discuss that any further.

CUDA is used to perform *General-Purpose computing on Graphics Processing Units* (GPGPU) and has been available since the GPUs became more programmable [15]. Before the GPUs became more programmable, you had different hardware units located on the GPU which was a specialized unit to execute the different parts of the GPU pipeline. However this was shown to be not that optimal since the pipeline could be given an uneven workload, therefore the GPU hardware units became more general and programmable which removed this problem on the GPUs with an uneven workload. But this also caused a side-effect of enabling GPGPU, and has become much easier nowadays.

2.4.1 Development

When developing in CUDA C/C++, you define a so called *kernel* (Listing 2.1), which is essentially a function that should be executed on the GPU, and the syntax is very similar to the usual C/C++ syntax. The main difference is that you have the prefix *"__global__"*, *"__host__"* or *"__device__"* before the function, and this specifies whether the kernel should be available for both the CPU and the GPU to execute, or if only the CPU/GPU should have access to the function.

Listing 2.1: CUDA kernel

```
1  /**
2  * CUDA kernel executed on the GPU
3  **/
4  __global__ void testFunction(int *array){
5          int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
6          int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
7          int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
8          int i = id_y*Nx*Nz + id_z*Nx + id_x;
9          array[i] = i;
10 }
```

And when you call these kernels, you have to specify the number of threads and number of blocks you want to execute the function on the GPU (block and grid size), where the syntax is shown in Listing 2.2.

Listing 2.2: Calling a CUDA kernel

```
1        // Specifies a 1D layout of threads (Total of 10*1024 threads)
2        dim3 block1d(10);
3        dim3 grid1d(1024);
4
5        // Specifies a 2D layout of threads (Total of 10^2 * 32^2 threads)
6        dim3 block2d(10, 10);
7        dim3 grid2d(32, 32);
8
9        // Specifies a 3D layout of threads (Total of 10^3 * 8^3 threads)
10       dim3 block3d(10, 10, 10);
11       dim3 grid3d(8, 8, 8);
12
13       // Calls the kernel with a given number of threads
14       testFunction<<<block1d, grid1d>>>(pointer);
15       testFunction<<<block2d, grid2d>>>(pointer);
16       testFunction<<<block3d, grid3d>>>(pointer);
```

As you may have noticed, there are some new terms used like *block* and *grid*, which is used when calling a CUDA kernel. This is how threads are structured when executing a function on the GPU, and is either a 1, 2, or 3 dimensional structure. These structures are then used to determine which threads that are executed together, and also the threads that have access to the same memory.

Then when the CUDA kernel is executed, you can use the *gridDim*, *blockDim*, *blockIdx*, and *threadIdx* to get the dimensions and the indices of of the current block and thread in X, Y, and Z directions.

2.4.2 Memory

When developing in CUDA you have access to a lot of different memory locations which impact the performance of your application. When developing for the CPU you have one main memory, and to obtain good memory performance you only need to think about the cache utilization, but there are not different kind of memory locations that you have available. But this is quite different on the GPU, where you have access to the following memory locations with different properties listed in Table 2.1[12].

Table 2.1: CUDA memory types

Memory	Location	Cached	Access	Scope
Register	On-chip	No	Read/write	One thread
Local	Off-chip	Yes[13]	Read/write	One thread
Shared	On-chip	N/A	Read/write	Threads within a block
Global	Off-chip	Yes	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read[14]	All threads + host

The memory model is also displayed in Figure 2.21 , where it is more easier to visualize how the threads and blocks are divided, and the memory locations they

[12]Data from: `http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921`

[13]Cached in the Fermi architecture and newer

[14]CUDA 2.1 and previous

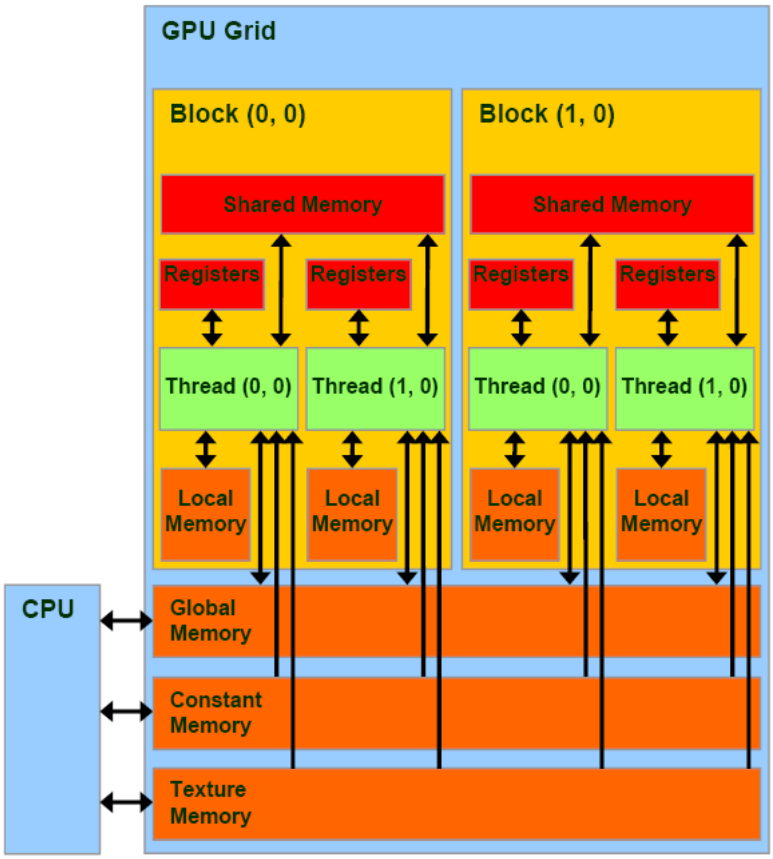


Figure 2.21: CUDA memory model, with permission from Nvidia

have access to. Also some of these memory locations are automatically used like registers and the local and global memory, however to take advantage of the fast on-chip fast memory, you have to explicitly declare certain variables to be located within the shared memory. It is also similar syntax when using the constant memory. However using the texture memory requires slightly more programming.

And regarding the actual performance of the different memory types, then the preferred memory types are the fast on-chip registers and shared memory. However these are extremely limited, e.g. on the Fermi architecture you have a total of 32768 registers per Streaming Multiprocessor (SM), which sounds like a lot of registers, but if you execute one block per SM with 1024 threads per block you end up with 32 registers per thread. And the size of the shared memory is actually configurable, where you have a total of 64 KiB per SM which is used for both the shared memory and a L1 cache.

2.4.3 Streaming Multiprocessor

The streaming multiprocessor (SM) is the basic building block of Nvidia GPUs that determines the GPU's performance. The SMs are changed from architecture to architecture, and consists of a given number of cores, special function units, load/store units and so on. A number of SMs are then added to the GPUs to increase the total number of cores available for the GPU.

In Figure 2.22 on page 29 the SM for the Fermi architecture is shown, where each SM has a number of 32 cores, a register file, 4 special function units, 16 load/store units, and some shared memory and caches. And within the SM you also have the *warp scheduler*, which is used to execute warps. Warps are in the

Figure 2.22: Streaming multiprocessor for the Fermi architecture

Nvidia terminology used for a fixed size group of 32 threads that are executed simultaneously, and Instructions are broadcasted to 32 CUDA cores within a SM which are used to execute 32 thread.

Since threads are executed simultaneously within warps, and the instruction are broadcasted to the group of threads, it is important to avoid any *thread divergence* when programming kernels where you need to branch. This mean that if your branching causes 16 threads within a warp to diverge from the other 16 threads, until the point where they join the same execution path, 16 threads will execute their instruction and the other 16 will perform a *no-op* i.e. become idle. Afterwards the other group of threads will perform theirs operation while the first group is idle. Then continue like this until the two groups has executed their set of instructions.

We will also look briefly at the SM in the Kepler architecture, which is called SMX in this architecture. The SMX is shown in Figure 2.23 on page 30 which has a total of 192 CUDA cores, 64 double precision units, 32 load/store units, 32 special function units, and twice as larger register file as in the Fermi architecture. And the warps scheduling system is also a more complex, where you have twice as many warp scheduler units.

Also since the streaming multiprocessor is the basic building block of the GPU, this gives the Kepler architecture a huge increase in the total number of cores available on GPUs. e.g the high end GTX-480 GPU with a number of 15 SMs gives a total of 480 CUDA cores, and the high end GTX-580 GPU with a number of 16 SMs gives a number of 512 CUDA cores. However when looking at the total number of CUDA cores on the Kepler architecture this gives a total of 2880 CUDA cores for the GTX-780 Ti, which has 15 SMXs.

Figure 2.23: Streaming multiprocessor for the Kepler architecture

2.4.4 GeForce and Tesla GPUs

The GPU industry are mainly driven by the demand of gaming, and it is only the last couple of years that we are using GPUs for scientific applications. And this transition has changed some of the demands of GPUs, and this is why you have two different types of GPUs namely the GeForce GPUs applicable for gamers, and the Tesla GPUs which is desirable when using GPUs for scientific applications.

The two types of GPUs are not that different, however there are a few noteworthy. Where the first most obvious difference is that there are no graphical output on the Tesla GPUs, which means that you have absolutely no way to connect a display to a Tesla GPU and use the device for rendering. The reason why this is removed is probably because it is removed in order to make room for additional hardware, which we will come back to. However, these kind of GPUs are used for calculations only, and if you need visualization you will have to connect e.g a GeForce GPU and transfer the memory between the two GPUs before you are able to visualize anything. But many scientific applications does not require any rendering, and utilizes the GPUs for computation only.

However, lacking the graphical output, Tesla GPUs will always have a considerably larger amount of memory available. e.g. the newest Tesla K40c has a total of 12 GiB of memory, and the GeForce GPUs usually has a maximum of 3-4 GiB. This is most likely caused due to that scientific applications can often require a lot of memory, and therefore it is very convenient to have a large amount of memory available on the GPU, such that you do not need to transfer parts of your data back and forth between the GPU device and the host main memory.

The Tesla GPUs also has additional hardware units used for double precision, which is sometimes required for scientific applications. e.g. the GeForce GPUs typically has a ratio of 1/24 between FP32 and FP64 units, and the Tesla GPUs has a ratio of 1/2 - 1/3.

Lastly, the Tesla GPUs have support for Error Checking & Correction (ECC) memory which is more discussed in Section 6.7.

Chapter 3

Previous & Related Work

In this Chapter, we will look at some previous and related work, where Section 3.1 is used to discuss the previous work, and Section 3.2 is used to look at some related work.

The previous work for this thesis include; the HPC-lab snow simulator, where we go into details for both the initialization and main loop of the snow simulator. Further we will look briefly at my specialization project which lead up to this master thesis, and lastly we will look into some work performed by Christian Sigrist, which performed fracture testing on different types of snow specimen, and it is from this work that I have obtained critical parameters for my simulations.

Regarding the related work we will first look at a Framework developed at *"WSL Institute for Snow and Avalanche Research SLF"*. This framework is used for modelling snow layers, and they also attempt to predict avalanches (Avalanche prediction is to my knowledge in an early stage. I looked into the API for the framework and found some method for retrieving a snow layer stability index, however how this is calculated is not know). We will also look into another project which uses the FEM method and fracture mechanics in 2D for avalanche prediction, however in 3D they uses another method called damage mechanics, where they predict avalanches based on the strain rate in the snow layers.

3.1 Previous Work

3.1.1 Snow Simulator

Previous work in the HPC-Lab at NTNU has been to develop a snow simulator, which simulates a given number of snow particles interacting with a wind field of a configurable resolution (see Figure 3.1 on page 33). And the snow simulator has been a focus area for several different master thesis and specialization projects, ranging from:

- Improving the visualization [16]
- Porting to GPU, using CUDA technology [2]
- Terrain Rendering Techniques[1]
- Optimal road routes generation using the A* algorithm [13]
- Porting to OpenCl[24] and OpenACC[14]

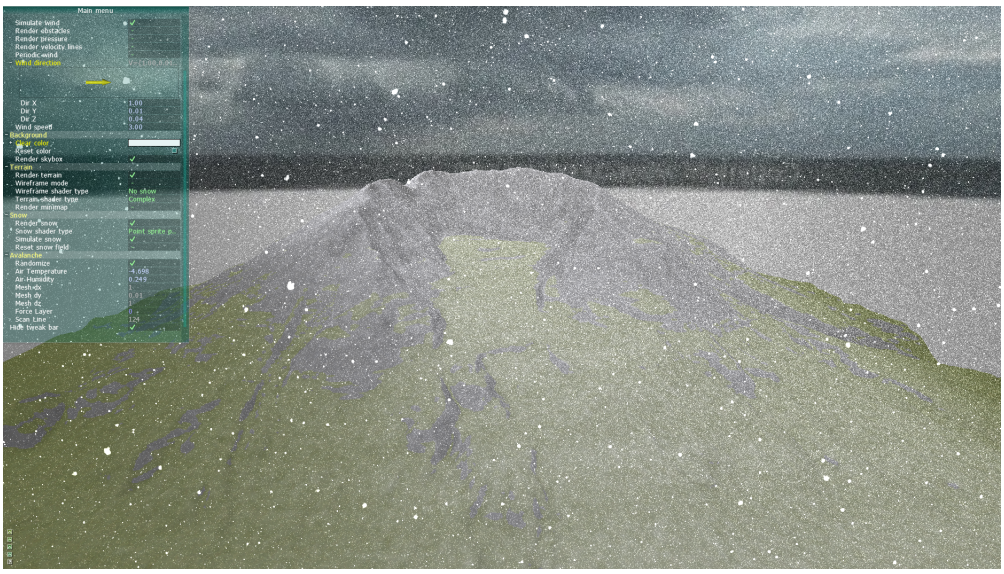


Figure 3.1: HPC-Lab snow simulator

- Snow layer modeling concept [26]

However, the initial snow simulator was developed by Ingar Saltvik in 2006[20], which could simulate 40 000 snow particles with 28 and 51 frames per second by using 1 and 2 threads respectively. Then later in 2009, Robin Eidissen ported the simulator over to CUDA[2] where he successfully simulated over 2 000 000 snow particles in real-time on a Nvidia GTX 280 GPU.

3.1.1.1 Initialization

The snow simulator starts up with a configuration screen which is loaded with default values. At this step the user can modify certain simulation parameters, and there is also some informational fields like; terrain map information and size, size of the window and fullscreen option, terrain shader type, wind field resolution in X, Y, and Z direction, shadow option, the number of snow particles, and a snow growth coefficient which determines how fast the snow cover grows. The configuration screen is shown in Figure 3.2 on page 34.

After the configuration is done, the user presses the "Start" button which starts the simulation based on the parameters specified in the configuration step. First the terrain is initialized by reading the terrain data which is a two-dimensional heightmap, and another 2D heightmap is also used for representing the snow cover height over the terrain. However, in the implementation, the two heightmaps representing the terrain height and snow height are added together into a single 2D array consisting of 4 float values, where the two first represents the X, Y coordinate, and the two latter represents the terrain and snow height at the X, Y coordinate. These heightmaps are then bound to a vertex buffer object such that OpenGL can be used for visualizing. The buffers are also associated with CUDA such that the GPU can update the snow cover data. Then all the terrain shaders are compiled and textures are loaded, and the snow layer modeling data is transferred to the GPU.

Afterwards the wind field is initialized, and in this step the pressure field used for simulating the wind field is initialized and the obstacle field is created. The obstacle field is a coarse version of the terrain data which the wind field uses,

Figure 3.2: Snow simulator configuration screen

and is periodically updated while the simulation progresses, due to the snow cover increases in height. The wind velocity data is also bound to the GPU as a texture, this is due to the interpolation in the wind field which depends on the spatial neighbours, and therefore the texture memory is more suitable to use due to the texture caching.

Then the last step of the initialization process is executed, and this is where the snow particles are initialized. First all the snow particles are initialized with a random location within the simulation domain, and they are also associated with a random spiral value $\exists\,[0, 2\pi]$ and determines the spiral flow of the snowflakes, the data is then bound to vertex buffer objects making the data available to OpenGL for rendering, and the vertex buffer is also associated with CUDA system, making it available for simulating the snow particles. Then lastly the shaders for the snow particles are compiled.

3.1.1.2 Main Loop

The main loop is the part of the simulator where all the calculations are performed. At this point, all the necessary data has been transferred to the GPU, and the CPU only instructs the GPU to perform several different tasks. In Figure 3.3 on page 35 is a short overview of the main loop of the snow simulator.

Firstly a minor operation is performed, which is calculating the number of frames per second (FPS). The FPS is also displayed while the simulator is running in order to have control over the original real-time requirement of the snow simulator.

After the FPS is calculated, the obstacle field is updated. However, since the obstacle field is only changing with increasing snow height, the update operation is only performed once every 1000 frame. Updating the obstacle field is performed on the CPU, and the algorithm loops over the 3D wind field and compares the obstacle field with the snow height data. The snow height data is transferred to

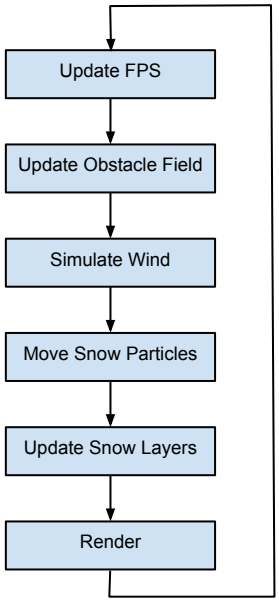


Figure 3.3: Snow simulator main loop

the host at the beginning of the function, then after comparison, the new obstacle field is transferred to the device.

After the obstacle field calculations are done, the simulation of the wind field is executed. The calculation of the wind field is performed on the GPU, and when the wind field is accessed the 3D texture memory is utilized due to the spatial access pattern in the wind field.

Next the snow particles are updated, and is also executed on the GPU, and uses the snow particles position in space, and the wind velocity field to calculate the motion of the snow particles. This part of the snow simulator is by far the most expensive kernel, and uses about 50% - 60% of the GPU time depending on the number of particles that are being simulated. Each particles also have a spiral value determining the amount of spin the particle should experience. If individual snow particles are observed in the snow simulator, we can see that they follow a circle path towards the terrain, and the radius of the circle is uniquely defined by each particle. Collision detection between the snow particles and the terrain is also performed at this stage, and since the terrain is represented by a 2D heightmap, this collision detection is very simple. And when a snow particle collides with the terrain, the particle is relocated to a random position in the top of the simulation domain, and the snow buildup is increased at the collision point with a configurable amount, and the snow buildup at the surrounding terrain neighbouring points are also increased with a given amount.

After the snow particles has been moved, and the snow buildup across the terrain has been incremented, the terrain snow layers are updated. At the previous version of the snow simulator[26] this consisted of two steps. Firstly, the snow layer model is updated, iff a new snow layer has been covered with snow, i.e. the snow has increased with a predefined amount of snow. (The snow layer model had the purpose of keeping track of the weather conditions present when each snow layer is filled, and was storing the air humidity and temperature). After the snow layer model is updated, some proof of concept calculations was performed in order to predict where avalanches could occur. This prediction was implemented in a scan line approach, due to huge mesh sizes, and visualization of the avalanche prediction

where implemented.

However, since the mesh generation process had to be changed in this project, the function for filling the snow layer mesh is no longer valid and will have to be reimplemented. This is discussed more in Section 7.3.1.

Then finally the render stage is started, where the render function for each part of the simulator is called. i.e. the snow class is responsible for rendering the snow particles based on the currently active shader, and similar for the terrain and wind classes. But the wind class is only rendering if certain debug options are selected, like rendering the wind velocity lines, or rendering the wind pressure field.

3.1.2 Snow Layer Modelling

In this project, we will continue the work of the previous *'HPC-lab Snow Simulator Improvement Terrain Model Expansion & Avalanche Prediction'*[26], but only parts will be used. The previous project consisted of two separate kernels, one which updated the snow layer model while the snow increases over the terrain, and another kernel which had the proof of concept calculations which predicted where avalanche could occur. However, these calculations where not physically correct, and in this project these calculations will be replaced by prediction based on fracture mechanics, which will be more physically correct.

In Figure 3.4, the snow layer model is visualized, where you have a configurable mesh size in 3D space. And in the figure you can see a set of red vertices and black vertices, representing the snow layers that are filled with snow. And for each vertex, properties are stored for representing the weather condition present when the vertex was covered with snow.

But again, since the mesh generation process had to be changed in this project, the algorithm for filling the mesh with weather condition data was no longer valid, however the mesh structure itself are still present in the code, and we simply have to rewrite the filling algorithm.

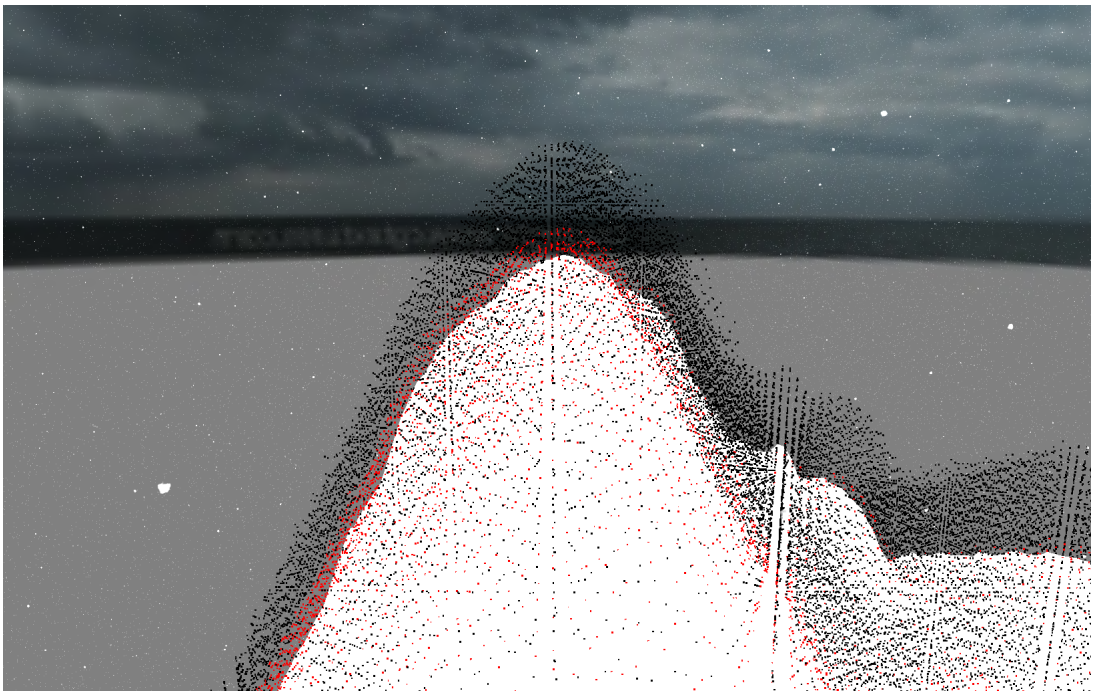


Figure 3.4: Snow layers over the terrain

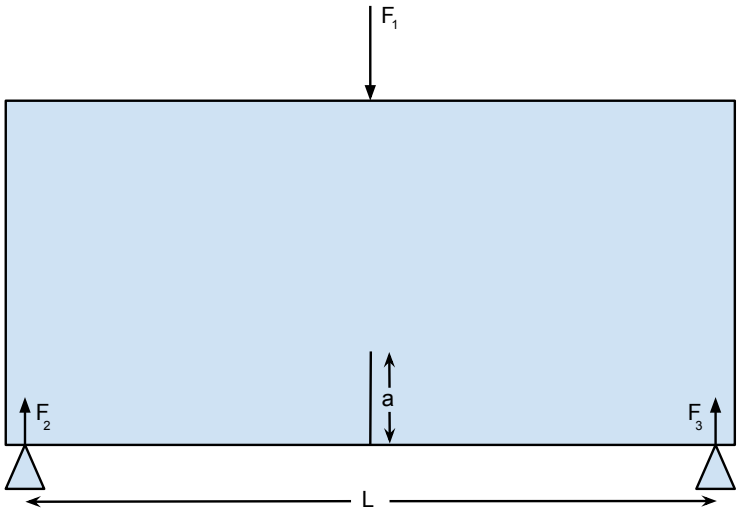


Figure 3.5: Three point bending test, $F_1 = 2F_2 = 2F_3$

3.1.3 Snow Layer Measurement

When using fracture mechanics to calculate where fracture will propagate in the snow layers, we will need to determine several different material constants for snow, and these constants are also varying for different types of snow. But there are some researchers around the world which carry out experiments on snow specimen, in order to discover these material constants. One of these researchers is Christian Sigrist from the Swiss federal institute of technology zurich, which performed different fracture experiments on different kind of snow specimen in his PhD thesis *'Measurement of Fracture Mechanical Properties of Snow and Application to Dry Snow Slab Avalanche Release'*[21].

In his thesis, he performed the *three point bending test* (3PB) and the *cantilever beam test* (CB). The 3PB test is shown in Figure 3.5, and is a method for measuring the tensile strength of a homogeneous material. When the load F is applied, then stress field within the specimen will form a mode I fracture, where the tensile force within the material is perpendicular to the fracture plane. The applied force and the displacement can then be recorded, in order to construct the force-displacement diagram and the stress-strain diagram, which plays an important role in fracture mechanics.

The force-displacement graph is a graph where the force is recorded while the object generating the force F_1 in Figure 3.5 is displaced a given amount. In Figure 3.6 on page 38 is an example of a load-displacement graph, where we can see that while the displacement increases, the force is increasing linearly up to a point where the structure fails. The stress-strain graph is also similar, however the Y-axis shows the stress $\sigma = F/A$, and the X-axis shows the strain $\epsilon = \Delta L/L$. Putting things more into perspective, when Sigrist performed the 3PB tests on snow specimen [21] the resulting graph is quite similar to the example in Figure 3.6, and the peak load of failure was equal to $F_f = 29.4N$, and the displacement at this point was $d = 1.2mm$.

The cantilever beam test was also used in order to find mechanical properties of heterogeneous snow, i.e. snow specimen consisting of multiple snow layers with different grain types. Especially specimen with a weak snow layer was tested, because it is strongly suggested that weak snow layers are an important factor

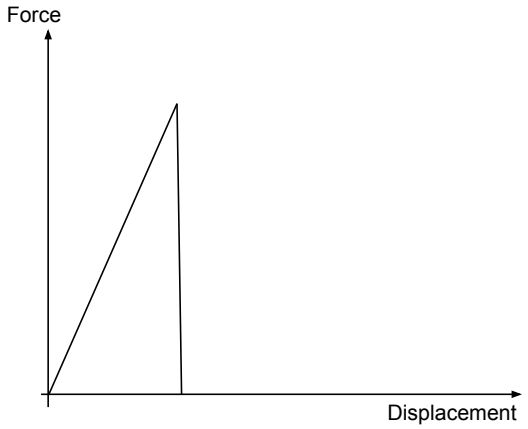


Figure 3.6: Example load displacement graph

when slab avalanches are released. The CB-test setup is shown in Figure 3.7 on page 39, where snow specimen consisting of three layers was tested. In the test, one layer L_2 is resting on a table, where another layer L_1 is only supported by the shear forces in the snow and a additional force is applied to the layer by the mass m, and between the two layers there is a weak layer with a cut of a given distance a, and the whole structure is also tilted a degree θ towards the table. A saw was then used to increase the distance a until failure.

Homogeneous snow tests: The tensile strength of the material was found by performing the 3PB test, and the critical stress intensity factor was also found in his experiments with mode I loading and found the following:

$$E \ \exists \ [14, 60] \ \ MPa \tag{3.1}$$

$$\sigma_f = 240 \left(\frac{\rho}{\rho_{ice}} \right)^{2.44} kPa \tag{3.2}$$

Where $\rho_{ice} = 917 kg/m^3$

and

$$K_{Ic} = 4.2 \times 10^{-4} \rho^{2.76} Pa\sqrt{m} \tag{3.3}$$

Heterogeneous snow tests:

$$E = 1.89 \rho^{2.94} Pa \tag{3.4}$$

$$\mathcal{G}_c = 0.044 \pm 0.020 J/m^2 \tag{3.5}$$

$$K_{II} = 0.49 \pm 0.36 kPa\sqrt{m} \tag{3.6}$$

Field tests:

$$\mathcal{G}_c = 0.066 \pm 0.014 J/m^2 \tag{3.7}$$

Critical energy release rate in a weak snow layer

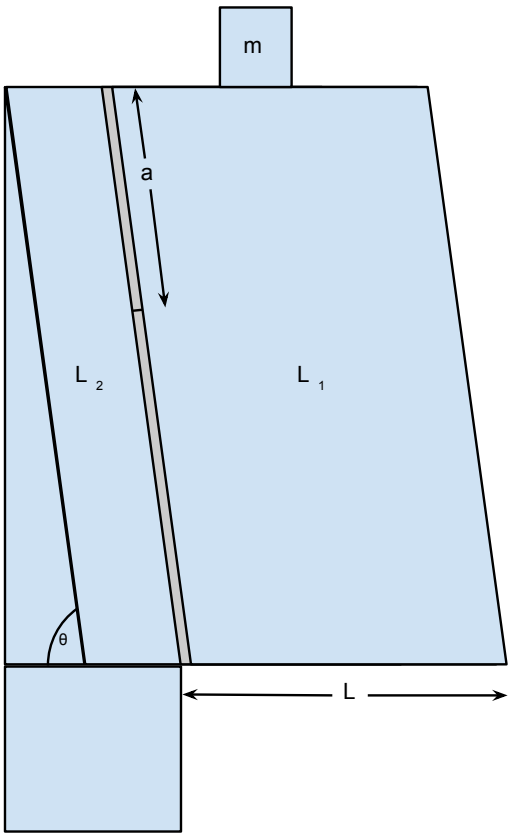


Figure 3.7: Cantilever beam test

3.2 Related Work

3.2.1 SNOWPACK

The institute for snow and avalanche research SLF in switzerland, has developed a model of the snow cover called SNOWPACK[3]. It models changes in the snow layer based on meteorological input and was primarily developed to improve the avalanche forecast warnings.

SNOWPACK is developed as a C library, and it solves the mass and energy balance equations by using a Finite Element numerical scheme, and the following individual processes are modelled in the snow layers:

- Heat Transfer
- Settling
- Phase change
- Water transport
- Metamorphism

It has also been written a lot of different reports of the SNOWPACK model, ranging from the development phase of the model where the numerical model and physical properties was the main focus [17, 11, 10], validation of the SNOWPACK model w.r.t actual snow layers[18], to avalanche forecasting by using the SNOWPACK model [5].

Hence SNOWPACK is a very active project, and they obtain very accurate modelling of snow cover. However, I do not know how the avalanche danger is calculated in the framework. Also, as stated earlier, the avalanche prediction in this library is in a quite early stage, and the home page[1] to the library has the following quote abote the avalanches prediction

'This is a first "shot" and it would be a miracle if we got it right at the very beginning.'

However, this framework would be perfect to model the snow layer properties and changes over time, which is an important factor when finding the critical energy release rate $\mathcal{G}_c$, but due to the time scope of this project the critical energy release rate had to be hard coded into the simulator, and when simulating different snow layers, the code has been changed and recompiled.

3.2.2 Snow Modelling

Martin Stoffel used the finite element method to model snow layers and avalanche formation in both 2D and 3D in 2005[22]. The 2D finite element model uses fracture mechanics in order to determine the fracture behaviour in these so called *weak-zones*, which is described in Section 2.2.2, under Paragraph *Slab avalanches:*.

However he states that the 2D model is difficult to apply in practical situations, because in his simulation, the initial conditions contains a weak-zone with a given lenght a, and in practice, the formation of these weak-zone are not known. And the 2D simulation only serves as a verification to the weak-zone model, which states that a precondition for a slab avalanche to occur are these weak-zones with a length between $6m < a < 10m$.

In the 3D model, he changes approach where he uses *Damage Mechanics*, which is suitable for making engineering predictions about the initiation and propagation of fractures within materials, without resorting to a microscopic description that would be too complex for practical engineering analysis. He also uses a so called *N-directional* approach, and is an idea of describing strain by using the normal strain in N evenly distributed directions, and can be modelled by simple spring elements.

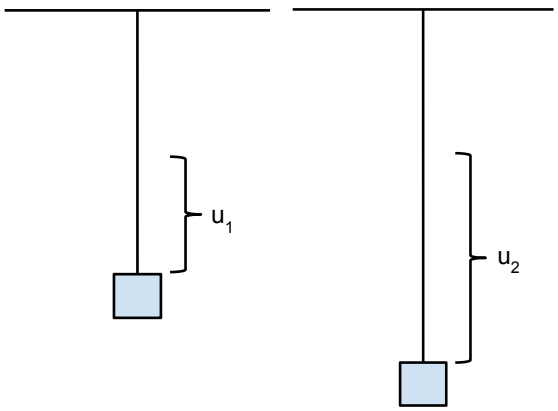
A simple example of how damage mechanics works is by Figure 3.8 on page 41, where a damage function is introduced into Hooke's law:

$$F = D(x) \times k\, u$$

where the damage function $D(x)$ return values in the range $[0, 1]$, this results that an undamaged spring has $D(x) = 1$, and thus the spring constant is equal to the original constant k, and a damaged spring gives $D(x) < 1$, resulting in $D(x) \times k < k$, which gives the result in Figure 3.8.

When having an undamaged spring with an external force applied at the end of the spring, it will result in a displacement u_1 (Figure 3.8a), and when the spring is damaged the end of the spring will be displaced by a greater value u_2, when the same force is applied (Figure 3.8b).

[1] `https://models.slf.ch/docserver/snowpack/html/index.html`



(a) Undamaged spring (b) Damaged spring

Figure 3.8: Impact of damage on elastic spring

This is then applied in his 3D avalanche model when calculating the strain rate of the snow. And it is also experimentally tested that when the strain rate is greater than approximately $10^{-4}s^{-1}$ snow behaves brittle, this is then used when calculating the damage function of the finite elements.

Chapter 4

Implementation

In this chapter, we will look at the implementation of this project, like the calculation of the displacement of the snow structure based on Hooke's law, and further how we derive normal and shear strain and stresses from the displacement.

We will also look into how we defined a relationship between the stresses and fractures, and then further how these stresses are used to calculate the energy release rate, which is necessary in order to calculate the propagation of fractures.

It should also be noted that the implementation discussed in this chapter was not the original approach that we had in mind, but is the result after a first attempt and is shown in Appendix G.

4.1 FEM

As described in Section 3.1.1.1, the snow simulator represents the terrain by a two dimensional heightmap, where a unique height value is stored at each (x, y) coordinate. When deciding the type of finite element being used for calculating the displacement field variable in the simulator, the first thought was parallelepiped across the entire domain. This would be the easiest mesh to generate, because the terrain vertices could be duplicated and add an additional increment dy in the Y direction, and then repeat the process a number of times. However, this kind of element would probably not generate any shear forces when all nodes are applied a force in Y direction only (see Appendix B for more description). And therefore a slightly more complex mesh generation was selected which results in a number of cubes across the entire domain with sides dx, dy, and dz, and this results that cubes located in steep slopes will experience more shear stress than cubes located in flat terrain.

However, the cube approach had another potential drawback, which is that the 4 vertices representing each side will not necessarily be within the same plane, and therefore the finite elements would not be a perfect cube. And how this would impact the simulation was not known, but a solution to this problem that was tested was to further subdivided each cube into 5 tetrahedrons. But after this method was chosen it was discovered that the displacement calculations was experiencing a vibration across the entire domain, and by testing the original finite element cube it was discovered that the vibration was slightly minimized. Therefore the cubes was selected as the finite element, and this is discussed more in details in Section 6.9.

In Figure 4.1 on page 43, the selected finite elements is displayed, and to solve the fracturing problem, we will first have to calculate the stress in the structure,

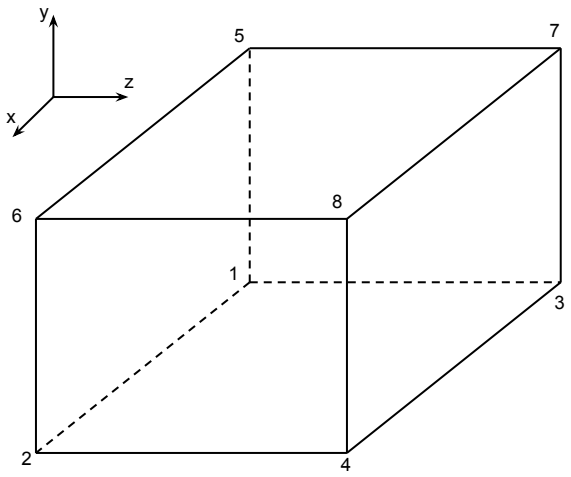


Figure 4.1: Finite element cube

which is calculated by the following steps;

First we need to calculate the displacement for each node in the structure, the displacement is solved by considering each element as a elastic material, and the displacement u can then be solved by the following equation when external load is applied (elaborated in Section 4.1.2):

$$f = ku \tag{4.1}$$

where $f \exists \mathbb{R}^3$ is the external force, $k \exists \mathbb{R}$ is the spring constant, and $u \exists \mathbb{R}^3$ is the displacement

Next we need to calculate the strain, which is defined as the displacement between particles divided by a reference length. The strain can be divided into normal and shear strain, and for a isotropic material which obeys Hooke's law, then normal strain is defined as the following:

$$\epsilon_x = \frac{\partial u_x}{\partial x} \tag{4.2a}$$

$$\epsilon_y = \frac{\partial u_y}{\partial y} \tag{4.2b}$$

$$\epsilon_z = \frac{\partial u_z}{\partial z} \tag{4.2c}$$

where ϵ is the strain, and u is the displacement

And the shear strain is defined as the following:

$$\epsilon_{xy} = \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} \tag{4.3a}$$

$$\epsilon_{yz} = \frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y} \tag{4.3b}$$

$$\epsilon_{xz} = \frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x} \tag{4.3c}$$

where ϵ_{xx} is the shear strain in the xx plane

Finally the normal and shear stress can be calculated by the definition of young's modulus:

$$\sigma = E\epsilon \tag{4.4}$$

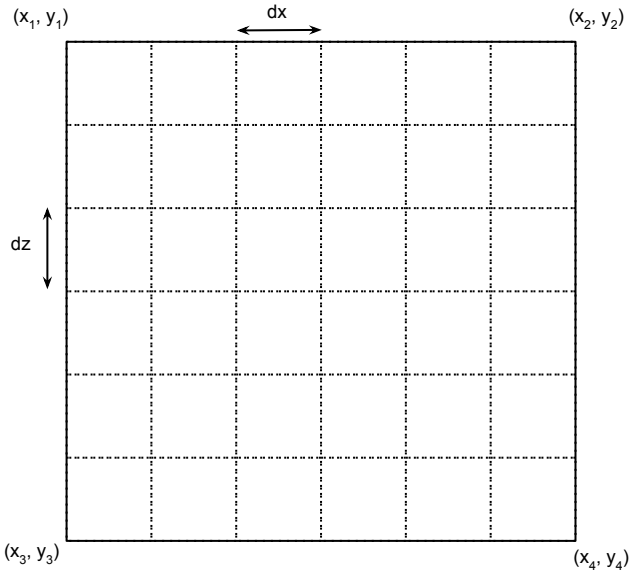


Figure 4.2: FEM mesh generation

where E is young's modulus, which is a material constant

To represent the snow layers as an elastic material may not be that accurate, however since snow is an extremely complex material, accurately modelling the snow layer mechanical properties would take a lot of time, and is therefore left out from this project. The main goal will be to calculate the fracture propagation based on the stresses in the structure, which again are depending on the displacement in the snow layers, and the displacement calculation can be substituted at a later stage to obtain more accurate results.

4.1.1 Mesh Generation

The size of the mesh determines the load on the GPU, therefore, the mesh generation process is implemented with a configurable dx, dy, and dz, such that the mesh can be easily reconfigured if compute or memory should be a bottleneck with the GPU used.

When generating the mesh, the first step is to generate all nodes located in the bottom layer. This layer will be generated based on the terrain vertices, and the mesh dx and dz. In Figure 4.2 the terrain heightmap is illustrated, where the height value stored in each node (x_i, y_i) vary across the entire terrain. The dx and dz is then used to either generate a finer or coarser mesh. And if $dx = dz = 1$, the bottom mesh layer will have exactly as many vertices as the terrain.

After the first layer of nodes is generated, the remaining layers are calculated based on the normal vector on the layer beneath the current, illustrated in Figure 4.3 on page 45. To calculate the normal vector at the terrain, we need 5 vertices at the layer underneath, the normal vector at (x, z) in the terrain will then be calculated as below:

$$\vec{n} = (v_1 \times v_2) + (v_2 \times v_3) + (v_3 \times v_4) + (v_4 \times v_1)$$

where v_1, v_2, v_3 and v_4 are illustrated in Figure 4.3

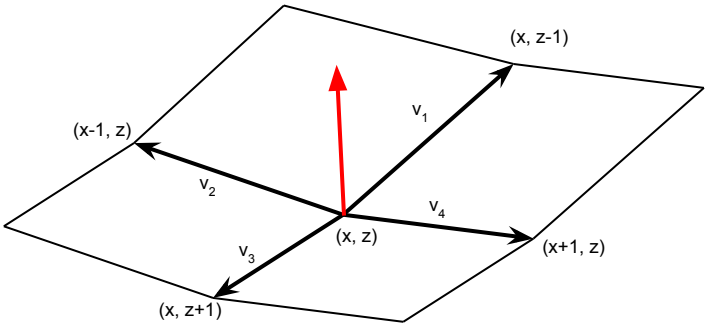


Figure 4.3: Normal vector calculation

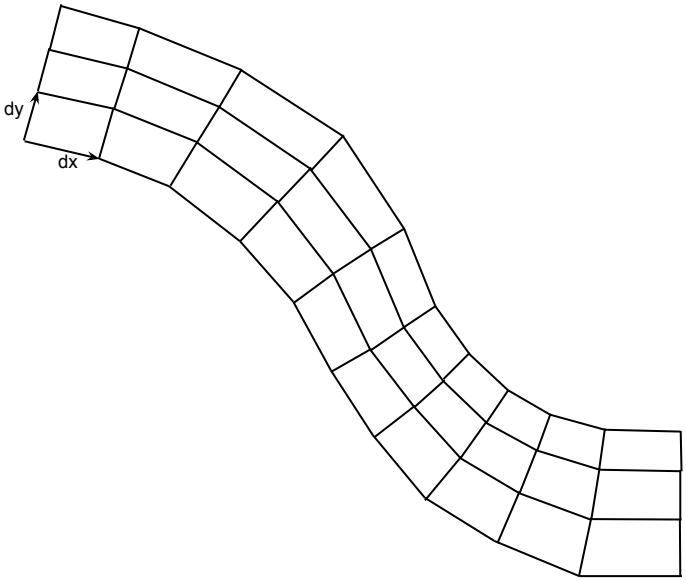


Figure 4.4: FEM mesh generation

The terrain normal vector is then normalized, then given the lenght dy which gives us the final position for the above node. The resulting mesh for the entire structure will then be as illustrated in Figure 4.4 on page 45.

4.1.2 Global Displacement

The calculation of the displacement of each node will be performed using a global numbering scheme of the nodes. The numbering is displayed in Figure 4.5 for the bottom mesh layer, and in Figure 4.6 on page 47 for a finite element number i, where M_x is the total number of nodes in X direction, and M_z is the total number of nodes in Z direction.

The relationship between the force and the displacement for each node will be; the spring constant times, the sum of the displacement for the neighbouring nodes in X,Y, and Z direction, minus the displacement of the current node:

$$F_j = k^{(i)} \left(U_x + U_y + U_z - 3U_j \right) \tag{4.5}$$

where F_j is the external force on node j, $k^{(i)}$ is the spring constant for a finite element number i, and U_j is the displacement of node j

If we then define these equations for all nodes for a given finite element i, and

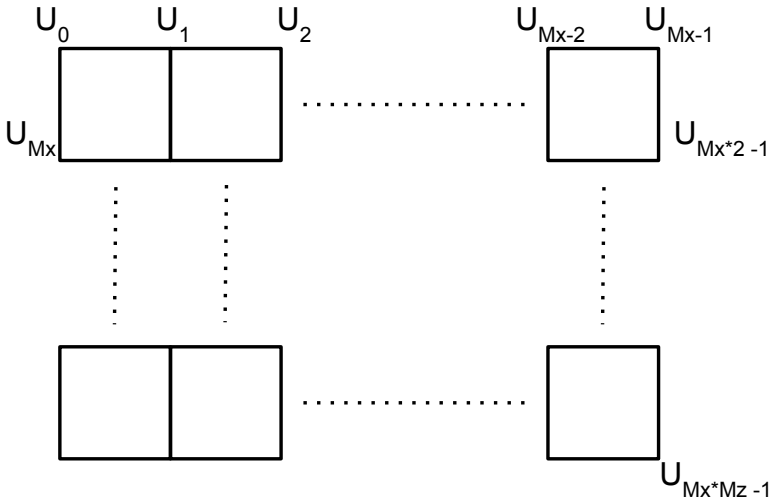


Figure 4.5: Global displacement numbering for bottom mesh layer

assemble these into a set of equations we will end up with the following system of equations:

$$k^{(i)} \begin{bmatrix} -3 & 1 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & 0 & 0 \\ 1 & -3 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \dots & -3 & 1 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 1 & \dots & 1 & -3 & \dots & 0 & 0 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \dots & 0 & 0 & \dots & -3 & 1 & \dots & 1 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 1 & -3 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & -3 & 1 \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 1 & -3 \end{bmatrix} \begin{bmatrix} U_i \\ U_{i+1} \\ \vdots \\ U_{i+M_x} \\ U_{i+M_x+1} \\ \vdots \\ U_{i+M_x*M_z} \\ U_{i+M_x*M_z+1} \\ \vdots \\ U_{i+M_x*M_z+M_x} \\ U_{i+M_x*M_z+M_x+1} \end{bmatrix} = \begin{bmatrix} F_i \\ F_{i+1} \\ \vdots \\ F_{i+M_x} \\ F_{i+M_x+1} \\ \vdots \\ F_{i+M_x*M_z} \\ F_{i+M_x*M_z+1} \\ \vdots \\ F_{i+M_x*M_z+M_x} \\ F_{i+M_x*M_z+M_x+1} \end{bmatrix} \tag{4.6}$$

And we can see that the stiffness matrix is symmetric, which is a requirement of the finite element method, and in addition the system of equation can be reduced to a 8×8 matrix because each finite element has 8 nodes, and 8 degrees of freedom. But since the matrix is consisting of vectors $\exists\, \mathbb{R}^3$, the stiffness matrix are in reality a 24×24 matrix, with 24 degrees of freedom. The systems of equations is solved by using the iterative method called *'Successive over-relaxation'*, with an initial guess of $x = 0$. The method it outlined in Listing 4.1[1].

Listing 4.1: Successive over-relaxation Method

```
1  Inputs: A, b, ω
2  Output: φ
3
4  Choose an initial guess φ to the solution
5
6  repeat until convergence
7      for i from 1 until n do
8          σ ← 0
9          for j from 1 until n do
10             if j ≠ i then
11                 σ ← σ + a_ij φ_j
12             end if
13         end (j-loop)
```

[1]Pseudocode obtained from `http://en.wikipedia.org/wiki/Successive_over-relaxation`

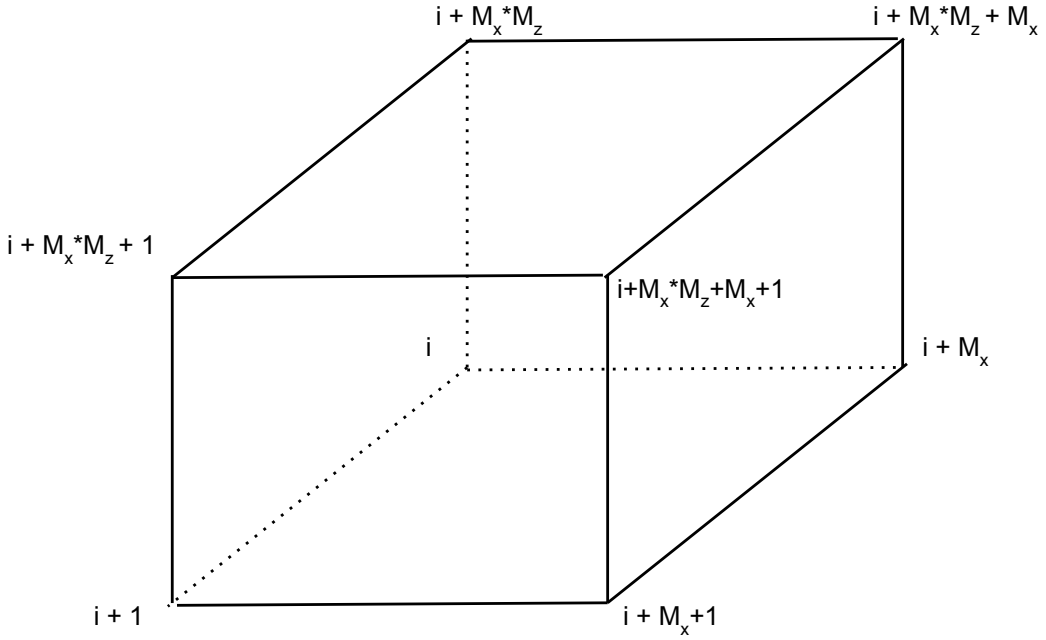


Figure 4.6: Global displacement for a finite element number i

```
14          φ_i ← (1 − ω)φ_i + (ω/a_ii)(b_i − σ)
15      end (i-loop)
16      check if convergence is reached
17  end (repeat)
```

And lastly we must add the constraint that the nodes located at the bottom layer will not have any displacement, i.e. $U_i = 0 \,\forall\, i \,\exists\, [0, M_x * M_z - 1] \subset \mathbb{N}$, to eliminate any rigid body motion of the entire structure.

4.1.3 Local Displacement

A fracture is the physical response from a system in a non-equilibrium state to achieve equilibrium. In our case, each element is experiencing a displacement of each node, resulting in a stress field within the structure, and when local stress concentrations are large enough, the stress can generate fractures which is the response of the structure to achieve equilibrium, i.e. fractures is generated in order to reduce the stress and displacement within the structure.

In order to define the fracture process as a process to obtain equilibrium, the process must decrease the displacement field within the elements with an increment in fracture length. This is accomplished by defining a relationship between a local displacement, the global displacement, and the fracture length a illustrated in Figure 4.7. The illustration shows that when a fracture occurs on a given plane between the finite elements, then the local displacements for the elements which are perpendicular to the fracture plane will be reduced by the following equation.

$$u_i = f\,(a)\, U_{g(i)} \tag{4.7}$$

where $f\,(a)$ is a function where $f\,(0) = 1$ and $\lim_{a\to\infty} f(a) = 0$, u is the local displacement, U is the global displacement, and $g\,(i)$ is a mapping function from local to global indices.

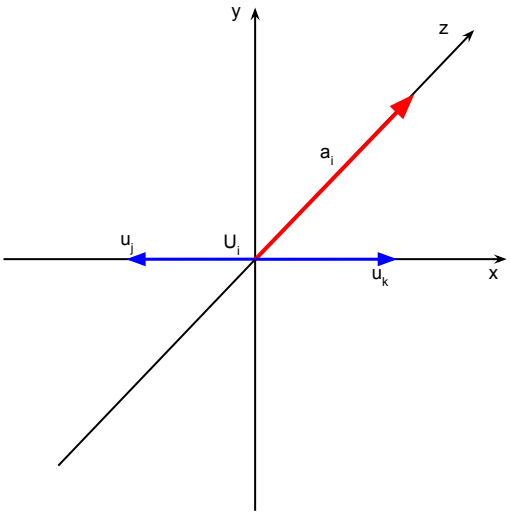


Figure 4.7: Fracture decreasing local displacement

The local displacements are then used when calculating the stress field within the structure, and this gives us that when a fracture propagates, the local displacements will be reduced, which again reduces the stress field. The $f(a)$ function used in this project is the following:

$$f(a) = \frac{1}{1 + \gamma a} \tag{4.8}$$

where γ is a configurable stress release factor

4.1.4 Local Strain and Stress

The strain calculation will be performed for each edge between two nodes of the finite elements as shown in Figure 4.8 on page 49, where both the normal and shear strain will be calculated. To calculate the normal strain between two vertices v_1 and v_2 with a displacement u_1 and u_2 we will be using the following equation:

$$\epsilon_n = \frac{||u_1 - u_2|| \ \frac{(u_1 - u_2) * (v_1 - v_2)}{||u_1 - u_2|| \ ||v_1 - v_2||}}{||v_1 - v_2||} = \frac{(u_1 - u_2) * (v_1 - v_2)}{||v_1 - v_2||^2} \tag{4.9}$$

where the numerator is the dot product between two vectors times the total displacement, and the denominator is the length between vertices v_1 and v_2

Where Equation 4.9 gives us the strain in normal direction only, and any shear deformation is left out.

The shear strain is also calculated, and is defined as the change in angle between two lines. So when calculating the shear strain between two vertices v_1 and v_2 where they both have a displacement u_1 and u_2 the dot product formula between two vectors will be used:

$$\epsilon_s = acos \left(\frac{(v_1 - v_2) \cdot ((v_1 + u_1) - (v_2 + u_2))}{||(v_1 - v_2)|| \ ||(v_1 + u_1) - (v_2 + u_2)||} \right) \tag{4.10}$$

After the normal and shear strain is calculated we can then find the normal and shear stress by the *Elastic modulus* which is defined as the following:

$$E = \frac{\sigma}{\epsilon}$$

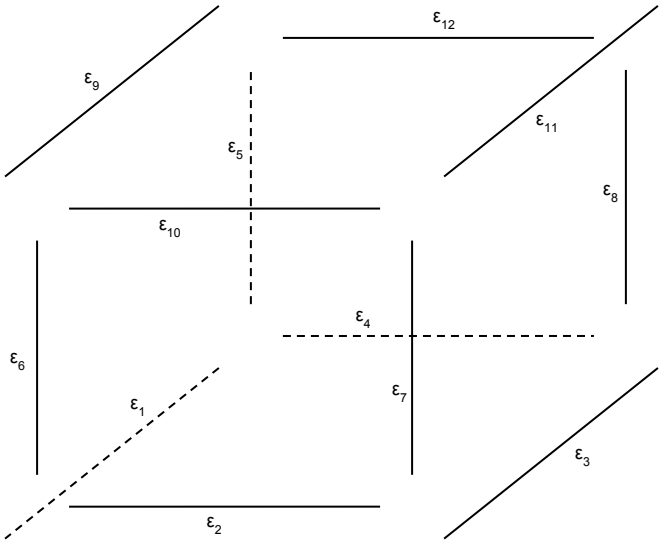


Figure 4.8: Strain calculation

4.1.5 Fracture Propagation

The last next step for calculating the fracture process can now be performed when the normal and shear stress is calculated. The method used is called *Elemental Crack Advance*, and calculates the energy release rate based on a specified *'da'* which represents the fracture increment.

However, according to [23] they state that this *da* value needs to be low in order to obtain accurate results, but they do not say anything about the magnitude of this parameter. And multiple simulations was therefore performed in order to obtain a value of *da*, and the results are shown in Section 5.2.1.3 and 5.2.1.4.

The fractures will be restricted to grow along the surfaces between the finite elements, and each node will have a vector containing data on the fracture length from that node, where each $a_i \exists \mathbb{R}^6$. This 6D vector can represent a fracture growing parallel with all axis, namly $\{\pm x, \pm y, \pm z\}$. By using these separate fracture length in both positive and negative axis directions, we can support fractures propagating independently in both directions.

If we look at Equation 2.15 on page 12, the energy release rate is defined for a elliptic fracture in a plate. However, this equation is defined for a fracture that propagates at the same rate on both sides of the element, i.e. the fracture length a is equal for $z = 0$ and $z = B$ (Figure 2.8 on page 12). But in this project the fractures will be able to propagate freely along all 3 axis, and therefore we will have formulate the energy release rate to better fit our elements, and David Roylance has a paper on introduction to fracture mechanics, where he gives a thoroughly description on the formula of calculating the energy release rate[19].

The energy release rate is calculated as the change in potential energy in the structure w.r.t. the fracture area. But the potential energy part of Equation 2.15:

$$\Pi_0 - \frac{\pi\sigma^2 a^2 B}{E}$$

is only valid for a fracture propagating in a plate with thickness B. The $\frac{\sigma^2}{E}$ part of the equation represents the strain energy per unit volume, and the rest of the equation, $\pi a^2 B$ represents an idealization of the volume where the strain energy is

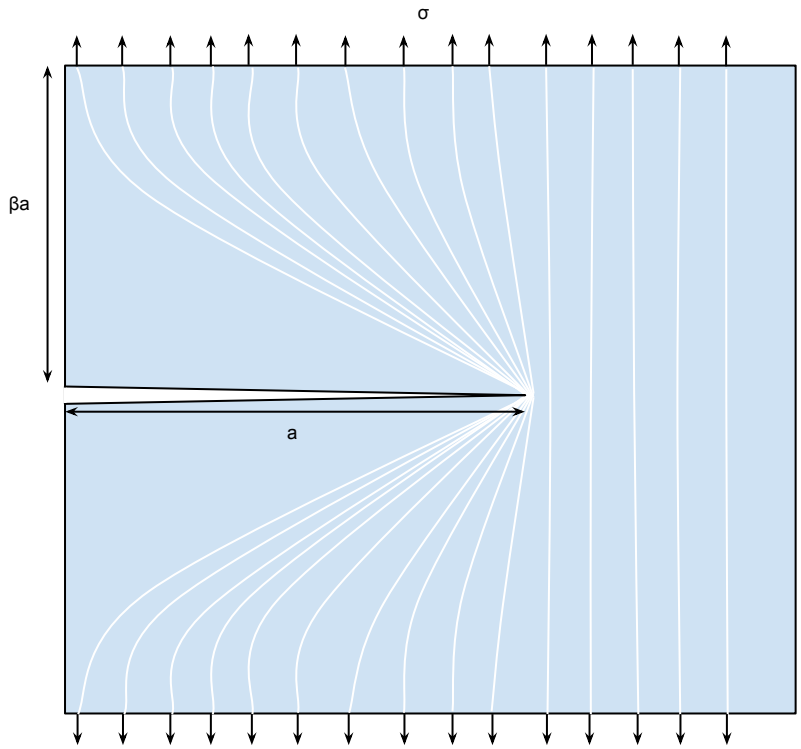


Figure 4.9: Idealization of energy unloading near fractures

released when a fracture propagates, and is shown in Figure 4.9 where two adjacent triangles with length a and height βa is unloaded. ($\beta = \pi$ for Inglis solution).

To adapt these calculations for our finite elements, where fractures can propagate individually we will have to calculate the volume of the tetrahedron represented by the length of the fracture vectors a_1 and a_2 shown in figure 4.10 on page 51, and the height of the tetrahedron will be $\beta \, ||a_1 + a_2||$. The total volume where the strain energy is released is then calculated by the following equation:

$$V = \frac{|| \, (\beta \tilde{a}) \, (a_1 * a_2) \, ||}{6}$$

where $\tilde{a} = ||a_1 + a_2||$

So when calculating the energy release rate for our finite elements, we need to solve the following equation (see Appendix C for derivation):

$$\mathcal{G} = -\frac{\frac{\sigma^2}{E} \, (V_i - V_e)}{A_e - A_i} \tag{4.11}$$

V_i is the initial strain energy volume without a fracture extension, A_i is the initial area of the fracture, and V_e and A_e is the end volume and area with a fracture propagation da of either a_1 or a_2

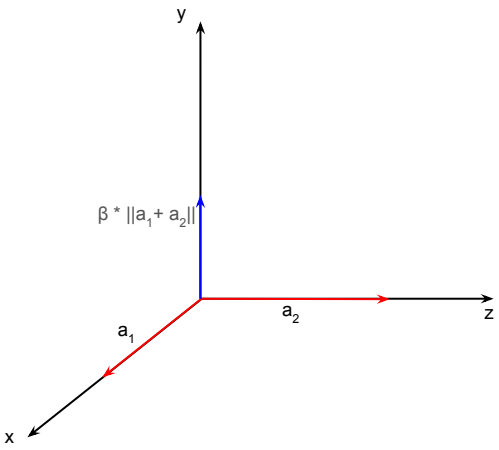


Figure 4.10: Volume of strain energy release w.r.t XZ plane

4.2 GPU Implementation

In the previous Section 4.1, we have look at the types of finite elements used in the calculations, how the mesh is generated, how we will calculate the global displacement of the structure, the relation between the global and the local displacement w.r.t. fracture length, calculation of the local stress, and finally how we need to change the calculation of the energy release rate in order to better fit our finite elements. In this section we will look into how this is implemented by using the CUDA technology available on Nvidia GPUs.

The first process is creating the snow layer mesh (described in Section 4.1.1), which is performed at the CPU in the initialization stage of the simulator. First the bottom layer of nodes are created, by simply duplicating the terrain vertices, but the mesh deltas are also taken into account, afterwards the rest of the snow layers are created based on the on the layer beneath. Code listings are shown in Appendix H.1 for initialization of the bottom layer, and the calculation of the remaining layers.

Next is solving the global displacement of the snow layers caused by external force acting on each mesh node (described in Section 4.1.2). This is solved by using the Successive over-relaxation method, which is an iterative method for solving a system of equations (Equation 4.6 on page 46). This part of the calculation and the remaining part is performed on the GPU, and when solving the global displacement problem, one iteration of the SOR method is performed every frame of the simulator. The GPU kernels are launched with the number of threads equal to the number of finite elements in the snow layers, and each thread will then have to solve the displacement of 8 nodes in total.

After the global displacement kernels are executed, the kernels which calculate the local strain, stress, and the energy release rate is executed and these kernsl also have a total number of threads equal to the number of finite elements. For each frame of the snow simulator, the kernels are launched as shown in Listing 4.2.

Listing 4.2: Kernels launched per frame

```
1  void CUDATerrainUpdate(){
2      ....
3      ....
4
5      // Init block and grid size for kernels
```

```
6       int x = 128;
7       int y = 1;
8       int z = 4;
9       dim3 grid(elements_x/x + 1, elements_y/y + 1, elements_z/z + 1);
10      dim3 block(x, y, z);
11
12      // Solving displacement
13      // vert = vertices
14      // U = global displacement
15      // F = global applied force
16      // mesh = snow layer properties
17      solve_global_displacement_step1<<<grid, block>>>(vert, U, F, mesh);
18      solve_global_displacement_step2<<<grid, block>>>(vert, U, F, mesh);
19      solve_global_displacement_step3<<<grid, block>>>(vert, U, F, mesh);
20      solve_global_displacement_step4<<<grid, block>>>(vert, U, F, mesh);
21      solve_global_displacement_step5<<<grid, block>>>(vert, U, F, mesh);
22      solve_global_displacement_step6<<<grid, block>>>(vert, U, F, mesh);
23      solve_global_displacement_step7<<<grid, block>>>(vert, U, F, mesh);
24      solve_global_displacement_step8<<<grid, block>>>(vert, U, F, mesh);
25
26      // Propagating fractures
27      // v = vertices
28      // U = global displacement
29      // frac = fracture lengths
30      // mesh = snow layer properties
31      // energy = color buffer used to visualize energy ratio (RGB)
32      // normal_stress = color buffer used to visualize normal stress (RGB)
33      // shear_stress = color buffer used to visualize shear stress (RGB)
34      // density = color buffer used to visualize density (RGB)
35      propagate_fractures_step1<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
36
37      propagate_fractures_step2<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
38
39      propagate_fractures_step3<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
40
41      propagate_fractures_step4<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
42
43      propagate_fractures_step5<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
44
45      propagate_fractures_step6<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
46
47      propagate_fractures_step7<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
48
49      propagate_fractures_step8<<<grid, block>>>(v, U, frac, mesh, energy,
            normal_stress, shear_stress, density);
50
51      ....
52      ....
53  }
```

4.2.1 CUDA Kernels

In this project, there have been a total of 16 kernels implemented in order to perform all the calculations necessary for calculating when and where fracture can propagate, however many of these kernels are very similar. First we have the kernels responsible for solving the global displacement of the snow layers based on a global applied force (self weight), which is divided into 8 kernels, namly:

- solve_global_displacement_step1<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step2<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step3<<< $grid, block$ >>>(vertices, U, F, mesh);

- solve_global_displacement_step4<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step5<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step6<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step7<<< $grid, block$ >>>(vertices, U, F, mesh);
- solve_global_displacement_step8<<< $grid, block$ >>>(vertices, U, F, mesh);

The above kernels are then launched for each finite element, and the step1 kernel will calculate the displacement of node 1 for each element, and step2 will calculate the displacement of node 2 for each element. And so on for the other kernels.

The reason why there are 8 different kernels for solving the global displacement is due to race conditions. At the beginning of this project, this was implemented as a single kernel which calculated the displacement for all 8 nodes at once, but since all the finite elements are sharing a lot of nodes, this lead to a lot of vibration in the structure. And when the calculation was separated into 8 kernels, the vibration was reduced dramatically.

One of the kernel for calculating the global displacement is shown in Appendix H.2 on page 149, and the only difference between these kernels are; the kernels calculating the displacement for the bottom 4 nodes has a *'if(id_ y > 0)'* check on line 21, which prevents any displacement for the bottom layer of nodes, and therefore removes any rigid body motion. And the other difference is indexing for both the displacement and force array.

The remaining 8 kernels are also divided similar, where they all perform calculation on their respective nodes on the finite elements, and these kernels are relatively large in comparison to the previous discussed. Where each of these kernels performs the following operations:

1. Find the Young's modulus and the critical energy release rate for the current finite element
2. Calculate the local displacement (Equation 4.7 and 4.8)
3. Read necessary vertices and displacement from memory, and calculate normal and shear stress (Equation 4.9 and 4.10)
4. Calculate the energy release rate (Equation 4.11)
5. If the energy release rate is larger than the critical energy release rate, extend the fracture a distance da

Step 4 and 5 above are then performed a total of 6 times, due to the fact that there are 3 fracture vectors that will exist in each node for a finite element, and each of these fracture vectors will lie within two different planes, where each plane could have different normal and shear stress.

One of the kernel for calculating the fracture propagation is shown in Appendix H.3 on page 150, and the other 7 kernels are similar with array indexing separating them.

Figure 4.11: Visualization methods

4.2.2 Visualization

In fracture mechanics, there are a lot of different fields variables which are important to visualize in order to get an in depth visualization of simulation. Therefore there have been implemented some visualization methods which can be changed in real-time from the main menu (see Figure 4.11), such that we can better examine the simulation.

All visualization methods (except *'Fracture length'*) uses a HSV coloring model to visualize the intensity of the field variable. Both the 'Value' and 'Saturation' of the HSV model is set equal to 1, and the 'Hue' is used to differentiate the intensity. The hue value is set equal to 240° for field values equal to zero, which gives a blue color, and when the field value increases, the hue value gets closer to zero, resulting in a red color. In Figure 4.12 on page 55 the range of colors with $S = 1$ is shown, and in addition we set $V = 1$ as stated, so the visualization will have the following color range with increasing field value; **blue**, **cyan**, **green**, **yellow**, **red**.

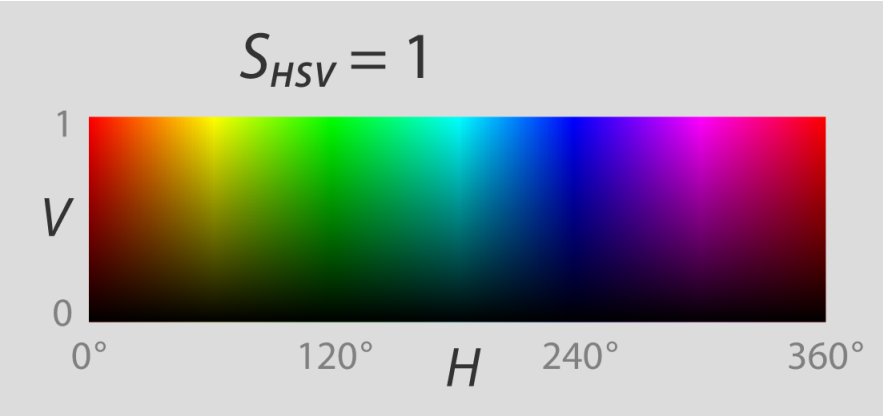


Figure 4.12: HSV coloring

Normal and Shear Stress: Stress is the main cause of any fractures, and is divided between normal and shear stress. Therefore the visualization of stress has also been divided into normal and shear stress visualization. The reason why the visualization has been divided, is that normal and shear stress will trigger fractures in different part of where avalanches are initiated, and therefore it is important to have the possibility of visualizing the different stress components. In Figure 4.13a and 4.13b on page 56 the normal and shear stress visualization is displayed respectively, where the force applied in the scenario is only along the Y-axis.

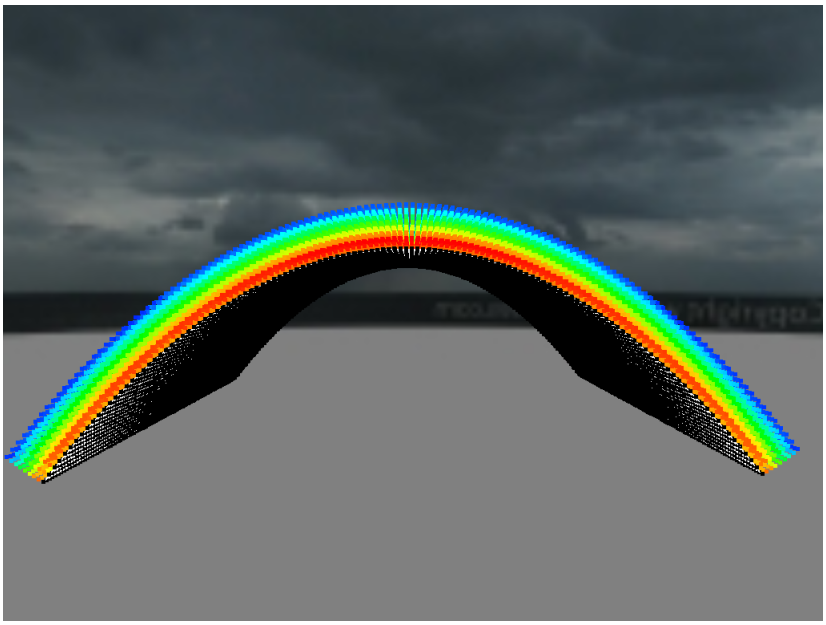
Energy Ratio: *'Energy ratio'* is a term used in this thesis when reffering to the ratio between the **energy release rate** and the **critical energy release rate**, and is another visualization method that are implemented in this project. And when this visualization method is activated, the exterior nodes of all the finite elements are visulized with their respective global displacement, and each vertex is given a HSV color of:

$$H = 240 - 240\frac{energy\ \ release\ \ rate}{critical\ \ energy\ \ release\ \ rate}, S = V = 1$$

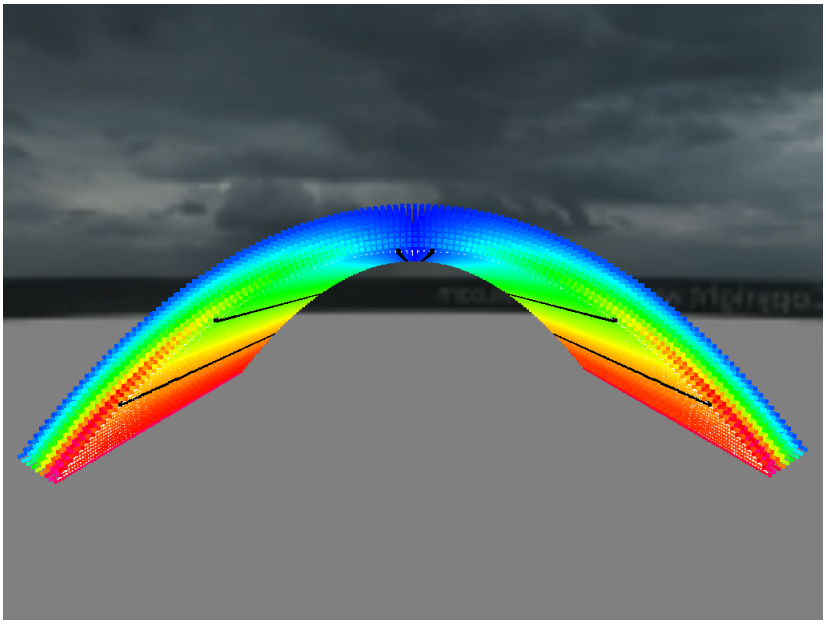
This results that each vertex will be red when there is a fracture process, and the fracture is extended a given distance da, and this visualization method is also be the best method to examine the snow structure to the point up to the fracture process, as we can see the color intensity increasing up towards fracture process. However, this method lacks visualization of the fracture lengths. In Figure 4.14 on page 57, the visualization method is shown and in this scenario, each nodes of the finite elements are applied a force in Y direction, resulting in the snow layers being compressed.

This visualization method method is also good for visualizing the stability of the snow structure, even when there are fractures occurring, because even if the fractures are extended a distance of da just once, it will most likely not occur any avalanches. This is because da value used is extremely low, in the range of nano / pico-meters.

Density: Density of the finite elements are also possible to visualize since the density has an impact on the critical energy release rate in the snow layers, which is stated in Equation 3.3 on page 38. The visualization method is displayed in Figure 4.15 on page 57, where the applied force is only along the Y-axis, causing an highest density at the top of the parabola.



(a) Normal stress visualization



(b) Shear stress visualization

Figure 4.13: Stress visualization

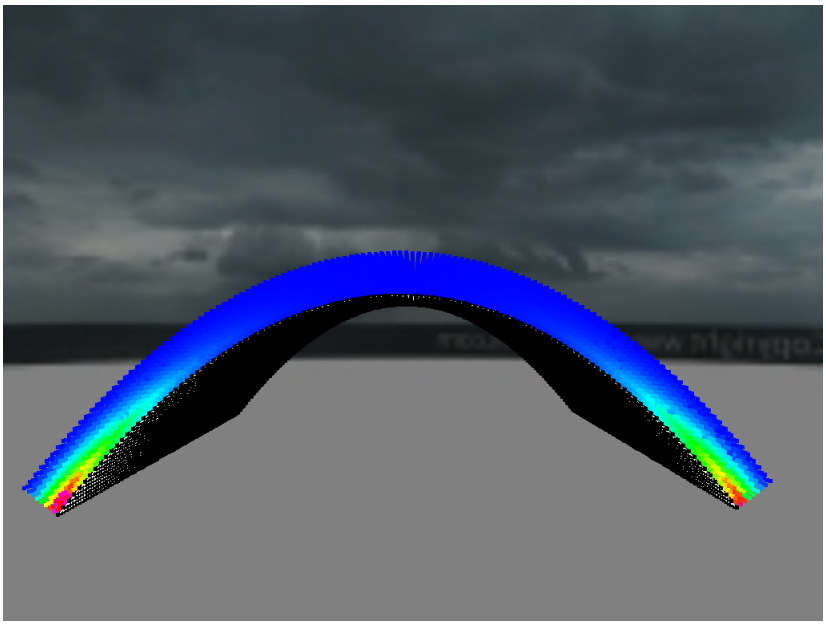


Figure 4.14: Energy release rate visualization method

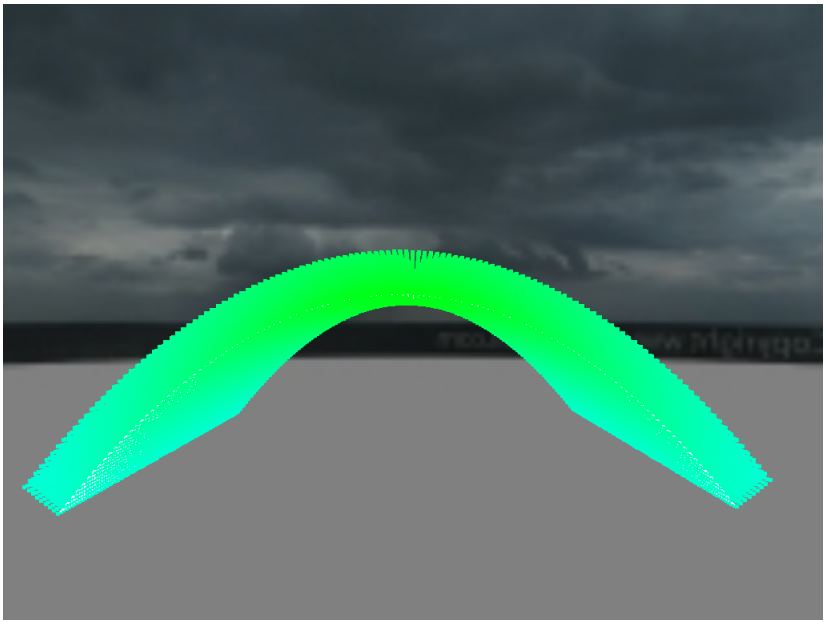


Figure 4.15: Density visualization method

Fracture length: Lastly the visualization of the fracture length is also added, since the other methods only provide a visualization of the finite element nodes, with a given color to represent the selected property. This method is focused on visualizing the lengths of each fractures present in each node of the finite element, which are able to grow independently of each other.

In Figure 4.16 on page 59, the visualization method is shown, where we can see a cross located in all the exterior nodes of the finite element. This cross represents the fracture vector $\exists\, \mathbb{R}^6$, and the cross is constructed the following way:

Listing 4.3: Cross construction

```
// Spatial center for exterior node
node = vertices[i] + displacement[i];

// X axis line
vertices[i*6*3 +  0] = node.x + fracture_length[i*6 + 0];
vertices[i*6*3 +  1] = node.y;
vertices[i*6*3 +  2] = node.z;
vertices[i*6*3 +  3] = node.x - fracture_length[i*6 + 1];
vertices[i*6*3 +  4] = node.y;
vertices[i*6*3 +  5] = node.z;

// Y axis line
vertices[i*6*3 +  6] = node.x;
vertices[i*6*3 +  7] = node.y + fracture_length[i*6 + 2];
vertices[i*6*3 +  8] = node.z;
vertices[i*6*3 +  9] = node.x;
vertices[i*6*3 + 10] = node.y - fracture_length[i*6 + 3];
vertices[i*6*3 + 11] = node.z;

// Z axis line
vertices[i*6*3 + 12] = node.x;
vertices[i*6*3 + 13] = node.y;
vertices[i*6*3 + 14] = node.z + fracture_length[i*6 + 4];
vertices[i*6*3 + 15] = node.x;
vertices[i*6*3 + 16] = node.y;
vertices[i*6*3 + 17] = node.z - fracture_length[i*6 + 5];
```

where *vertices* is the spatial location of each node, *displacement* is the node displacement, and *fracture_length* is the fracture vector $\exists\, \mathbb{R}^6$.

The code in Listing 4.3, will generate 6 vertices for each finite element node which are used for drawing 3 lines representing the fracture lengths. However, the disadvantage with this visualization method is that since the vertices are not explicitly stored, and only the length of the fractures are stored. These vertices has to be calculated for each frame of the visualization, which is executed on the CPU. i.e. when this method is activated the total performance of the simulator will drop.

4.2.3 Memory Requirement

In the beginning of this project, it was soon realised that the memory requirement for the simulation would be huge, and it increases rapidly with increasing mesh size. Most of the data is required for the visualization part, however this could been implemented differently to reduce the total memory requirement. e.g. all the color buffers used for the different visualization methods have their own dedicated buffer. Instead one buffer could be used, and populating this buffer according to the activated visualization method. But in the current implementation, all visualization methods have their own dedicated color buffer. But by using this approach, we can pause the simulation and switch between the different visualization methods without having the need of running one extra frame to update the color buffer.

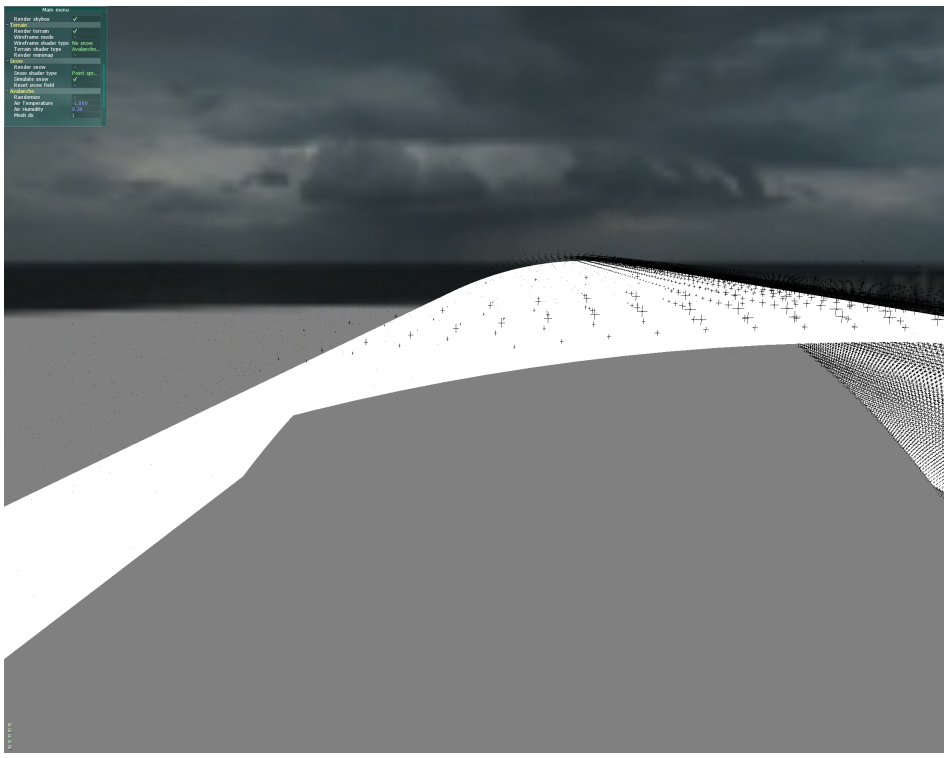


Figure 4.16: Fracture visualization method

However, the memory requirement in the implementation depends mainly on the number of nodes used for the finite elements, and the total number of finite elements. The total number of nodes are calculated by the following equation:

$$nodes = \left(\frac{(resolution - 1)}{dx} + 1\right) \times \left(\frac{(resolution - 1)}{dz} + 1\right) \times (num\_snow\_layers + 1)$$

And the total number of finite elements are calculated by the following equation:

$$elements = \left(\frac{(resolution - 1)}{dx}\right) \times \left(\frac{(resolution - 1)}{dz}\right) \times (num\_snow\_layers)$$

Where the number of snow layers are calculated by the following equation:

$$num\_snow\_layers = \frac{\frac{max\_snow\_height}{dy}}{\frac{SCENE\_Y}{resolution}} = \frac{max\_snow\_height \times resolution}{dy \times SCENE\_Y}$$

Resulting in the following memory requirement:

$$mem = 180nodes + elements \quad Bytes$$

This gives a total number nodes and memory requirement shown in Table 4.1 for a vararity of map sizes, and the other values in the equation for calculating the total number of nodes set equal to the following:

- Mesh deltas = $\{1.0, 1.0, 1.0\}$
- Max snow height = 4.0
- SCENE_Y = 64

Table 4.1: Total nodes for different terrain sizes

Resolution	Nodes	Elements	Memory (MiB)
100	70 000	58 806	12.07
150	225 000	199 809	38.81
200	520 000	475 212	89.71
300	1 710 000	1 609 218	295.08
500	8 000 000	7 719 031	1380.65

Chapter 5

Result

In this Chapter, we will perform various tests on the implemented simulator, where we first in Section 5.1, will look at the workstation setup regarding the hardware and software used. And we will also look briefly at the compilation step.

Next in Section 5.2, we will look at the test results, which is divided into *"Simulation Results"* and *"Performance Results"*. Where we in Section 5.2.1, will look at the simulation results like; normal and shear stress distribution, displacement calculation, and fracture propagation. And in Section 5.2.2, we will look at the performance results which includes; CUDA kernel analysis where the optimal configuration of threads and blocks are found, arithmetic and memory usage for the two main type of kernels, and other parameters which has an impact on the performance.

A CPU implementation has also been implemented in this project, where we compare the CPU versus the GPU. But it should be noted that due to time limitations, there has not been enough time to perform any code optimization.

It should also be noted that after all the testing was performed, it was discovered that ECC memory was disabled on workstation 1, and enabled on workstation 2. And this has an impact on the performance on memory bound applications. Therefore, all results obtained on workstation 1 is slightly better than they should be. The actual impact is looked more into details in Section 6.7, and it was discovered that the relativ impact decreases with the problem size.

5.1 Setup

In the following sections, we will look into the hardware of the workstations used for testing the performance of the simulation, and we will also describe the compilation and the necessary libraries that the simulator is depending on.

5.1.1 Compilation

For all workstations the same Makefile is used, and also the same libraries are necessary. When compiling the c++ code, the optimization level 3 is used, and the c++ standard used is *'-std=c++0x'*.

For all the CUDA code, the *-m64* flag is used to force the compiler to compile 64-bit code, and the flag *–use_fast_math* is also used in order to speed up any complex math functions by reducing the accuracy. We also specify the *-arch* flag on each workstation, depending on the actual hardware present. e.g workstation 1 uses *-arch=sm_20*, and workstation 2 uses *-arch=sm_35*.

Lastly, all object files are then linked together with the following libraries (full makefile avaiable in Appendix H.4):

- *-L/usr/local/cuda/lib64/*
- *-lcudart*
- *-lGL*
- *-lglfw*
- *-lGLEW*
- *-lGLU*
- *-lAntTweakBar*

In Table 5.1, the full overview of all dependencies of the simulator is shown, and was obtained by running the command *"ldd snow"* in linux.

Table 5.1: Snow simulator shared libraries dependencies

Library	File
Virtual dynamic shared object	linux-vdso.so.1
CUDA runtime library	libcudart.so.5.5
OpenGL	libGL.so.1
GLFW	libglfw.so.2
GLEW	libGLEW.so.1.6
AntTweakBar	libAntTweakBar.so.1
The GNU Standard C++ Library	libstdc++.so.6
C Math library	libm.so.6
GCC low-level runtime library	libgcc_s.so.1
Standard C library	libc.so.6
POSIX Threads library	libpthread.so.0
Dynamic linking library	libdl.so.2
POSIX.1b Realtime Extensions library	librt.so.1
Thread local storage support for the NVIDIA OpenGL libraries	libnvidia tls.so.331.20
OpenGL core library containg the core accelerated 3D functionality	libnvidia-glcore.so.331.20
Client interface to the X Window System	libX11.so.6
Common X Extensions library	libXext.so.6
X Resize and Rotate library	libXrandr.so.2
Dynamic loading library	ld-linux-x86-64.so.2
Iinterface to the X Window System protocol	libxcb.so.1
RENDER extension library	libXrender.so.1
X Authorization routines	libXau.so.6
X Display Manager Control Protocol	libXdmcp.so.6

5.1.2 Hardware

As stated, this project will be tested on a variety of different workstations, where the hardware and software will vary. In this section, we will look at the hardware and software present in these workstations. The specification of the workstations can be found in Table 5.2 and 5.3 on page 63.

Table 5.2: Workstation 1 specifications

Workstation 1: Hardware	
CPU	i7-3770 @ 3.40 GHz
GPU	Nvidia GTX-480 (1.50 GiB)
	Nvidia Tesla C2070 (6.00 GiB)
Memory	16 GB DDR3 1600 MHz
Power Supply	CORSAIR AX1200
Workstation 1: Software	
Operating System	Ubuntu 12.04 64 bit
Nvidia Driver	331.20
CUDA Toolkit	4.0
g++	4.6.3

Table 5.3: Workstation 2 specifications

Workstation 2: Hardware	
CPU	i7-4771 @ 3.50 GHz
GPU	Nvidia GTX-760 (4.00 GiB)
	Nvidia Tesla K40c (11.25 GiB)
Memory	32 GB DDR3 1600 MHz
Power Supply	CORSAIR AX850
Workstation 2: Software	
Operating System	Ubuntu 12.04 64 bit
Nvidia Driver	331.20
CUDA Toolkit	5.5
g++	4.6.3

5.2 Tests

In the following sections, we will first look at the simulation results, which is mainly consisting of a large set of simulation screenshots. Afterwards we will look at the performance results by changing the hardware being used for the simulation, and the problem size.

5.2.1 Simulation Results

In the following sections, we will look at the simulation results by changing different parameters, and look at how the different parameters impacts the simulation. The parameters we will vary are; the terrain, Young's modulus, the critical energy release rate, the fracture propagation distance *da*, and finally the relaxation factor used by the iterative *'Successive over-relaxation'* method.

The applied force and the spring constant is determining how much displacement the structure will obtain, and strain and stress is derived from the displacement. However, in our cases we will examine the snow under self weight, and therefore the global applied force is set as constant for all simulations.

The parameters which has the most impact on fracture propagation, are the following:

- Density
- Critical energy release rate
- Fracture propagation distance *da*
- Stress (discussed above)

The density of the snow is calculated by the size of the finite elements, which changes due to the displacement, and the actual weight of the finite element is set as constants. The critical energy release rate is utmost important, since it is the energy required to extend the fractures a given distance *da*, and this will vary while simulating stable and unstable snow. And it was found that the fracture propagation distance *da* have a big impact on the calculations.

5.2.1.1 Displacement Calculation

The first part of the simulation consists of calculating the displacement of vertices, which further generates internal stress. The displacement calculation is based on FEM, where Hooke's law is the governing equation. And as stated earlier, these equations are assembled into a large system of equations, which is solved by the iterative method *"Successive over-relaxation"*.

This method uses a relaxation factor ω, which is used to speed up the convergence. And when this relaxation factor $\omega > 1$, the method is called Successive over-relaxation, and when this factor is $0 < \omega < 1$, the method is called Successive under-relaxation. And usually a high relaxation factor is desirable to speed up the convergence rate.

In the testing process, it was then discovered that the SOR-method is not converging when $\omega > 1$, and when ω is approximately equal to one, the calculations are experiencing some instabilities, however these minor instabilities are scaled grately when calculating the stress, causing large stress variation.

The results are shown in Figure 5.1 on page 66, and when using $\omega = 0.1$ (Figure 5.1a), the displacement calculation is highly stable, resulting in a stable stress level. However, when using $\omega = 1$ (Figure 5.1b), we can observe minor vibration in the vertices, and this results in a highly unstable stress calculation. We can also see that the overall stress is higher when using $\omega = 1$.

Finally we can see that when using $\omega = 1.1$ (Figure 5.1c) the structure is literally ripped apart, and this process continues until all the vertices has a absolute value of their Y coordinate so large that the vertices are no longer visible. We can also see that the overall stress level is even higher.

5.2.1.2 Stress Distribution

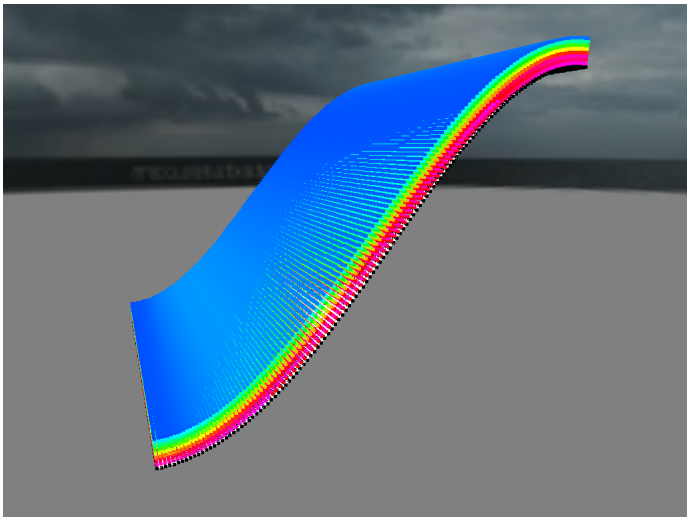
In this section, we will look into the stress calculations for a variety of different terrains, to verify that shear and normal stress calculations seems correct. The spring constant and Young's modulus will remain constant over the following tests, because they act as scalar values to increase the amount of stress in the structure, and in our visualization, all values need to be normalized before the visualization.

We will compare the amount of stress, and which type of stress that we find in the terrain, where we should expect increasing shear stress with increasing slope. For the following tests we use the following parameters:

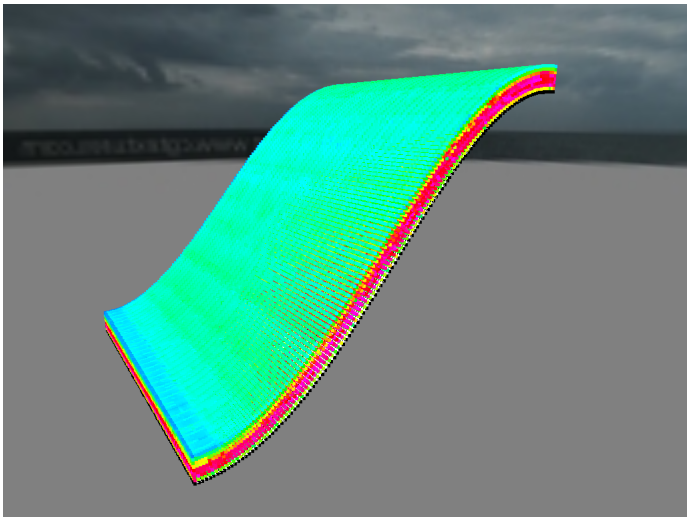
- Applied force $\vec{F} = \{0 \ \ 200 \ \ 0\}^T$
- Spring constant $k = 10000\rho$
- Finite element mass $m = 20kg$
- Young's modulus $E = 12MPa$
- Stress release factor $\gamma = 1000$ (Equation 4.8)
- $\omega = 0.1$

The stress intensity is normalized to $stress/10000$, and the results are shown in Figures 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 (page 67 to 72). But it should be noted that since we do not have any experimental data to compare our stress distribution results with.

But the result seems physical accurate, where we can find normal stress for all kinds of terrain. And the shear stress is only present where there are slopes, and increases as the slope increases. And also that both types of stress has the largest intensity towards the bottom of the snow layers.



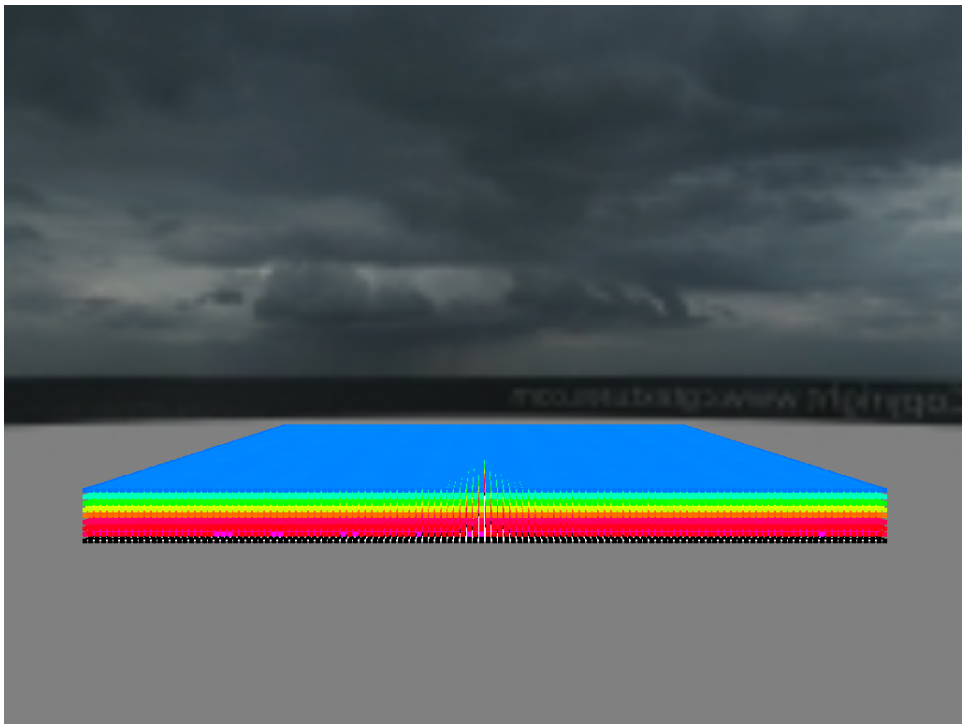
(a) $\omega = 0.1$



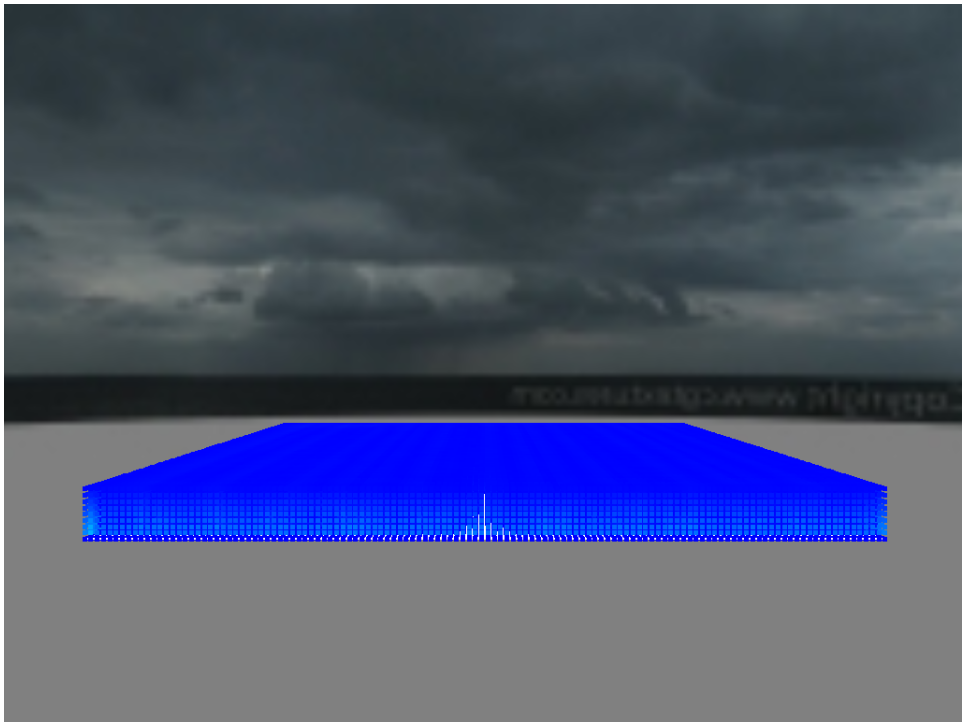
(b) $\omega = 1$

(c) $\omega = 1.1$

Figure 5.1: ω testing for displacement calculation

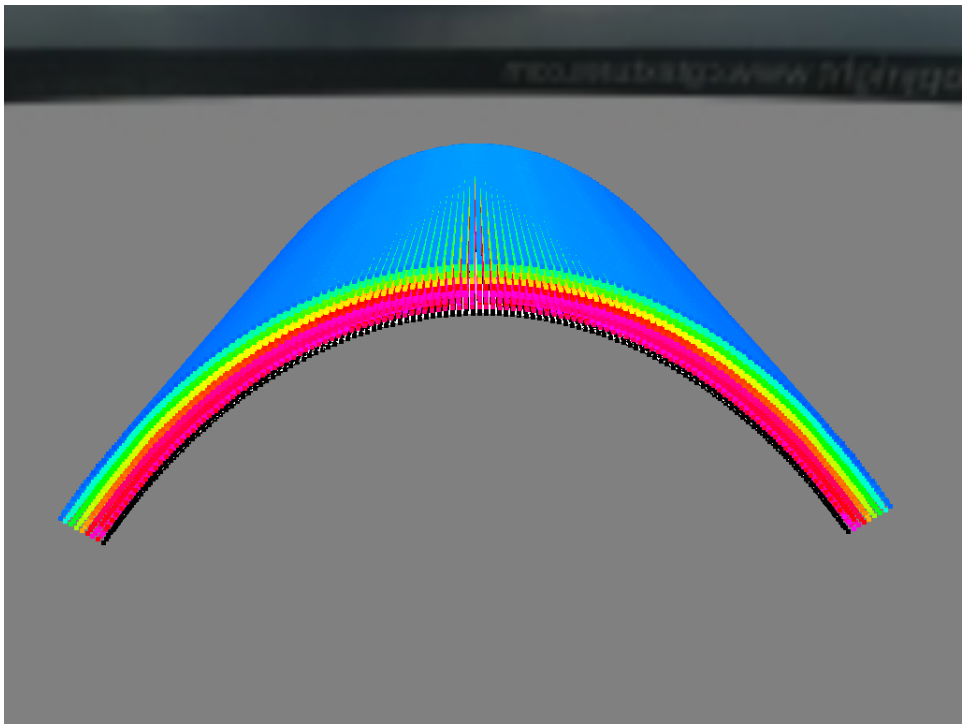


(a) Normal stess

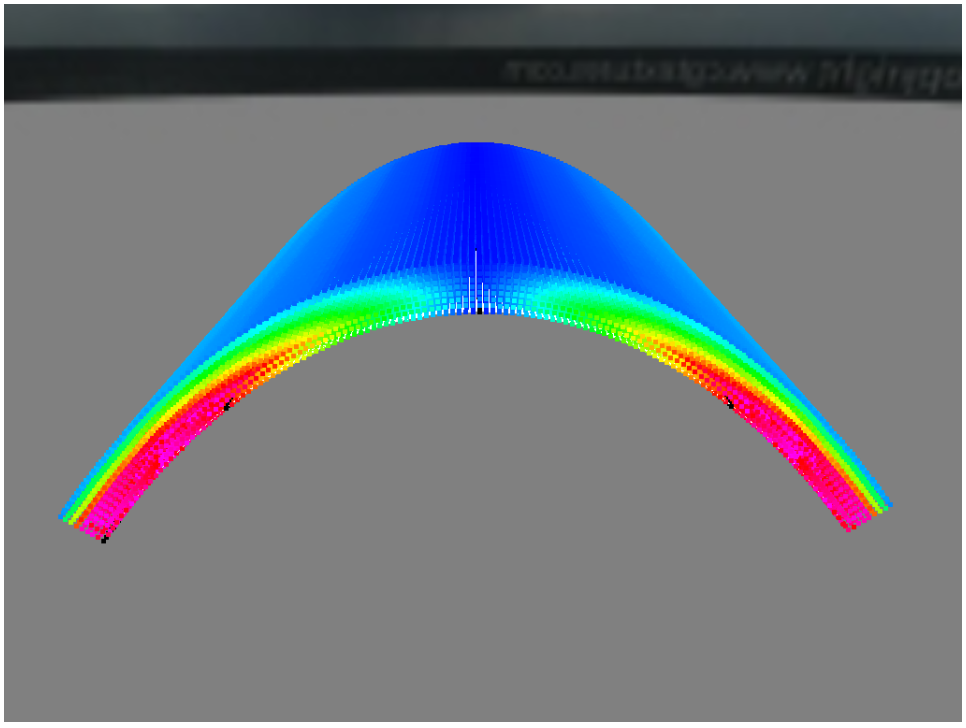


(b) Shear stress

Figure 5.2: Stress distribution for flat terrain

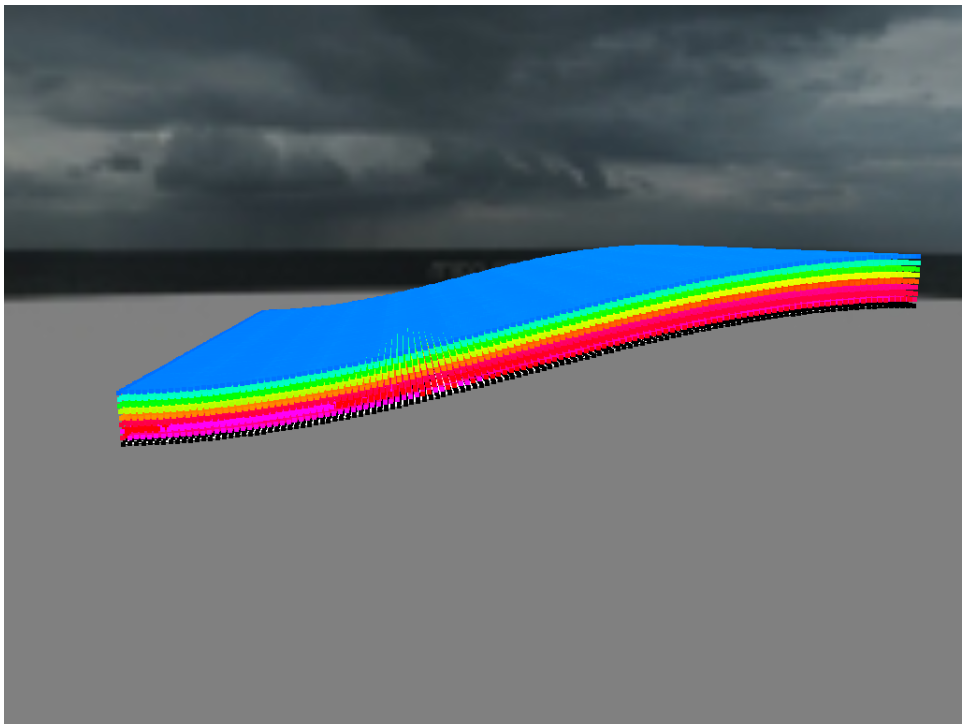


(a) Normal stress

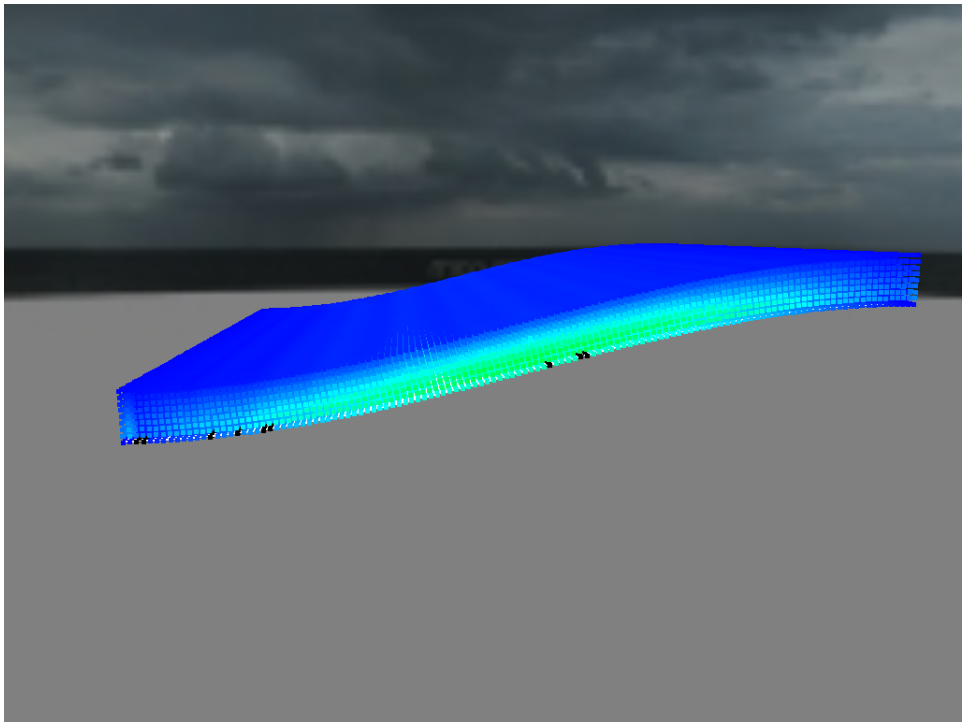


(b) Shear stress

Figure 5.3: Stress distribution for parabola terrain

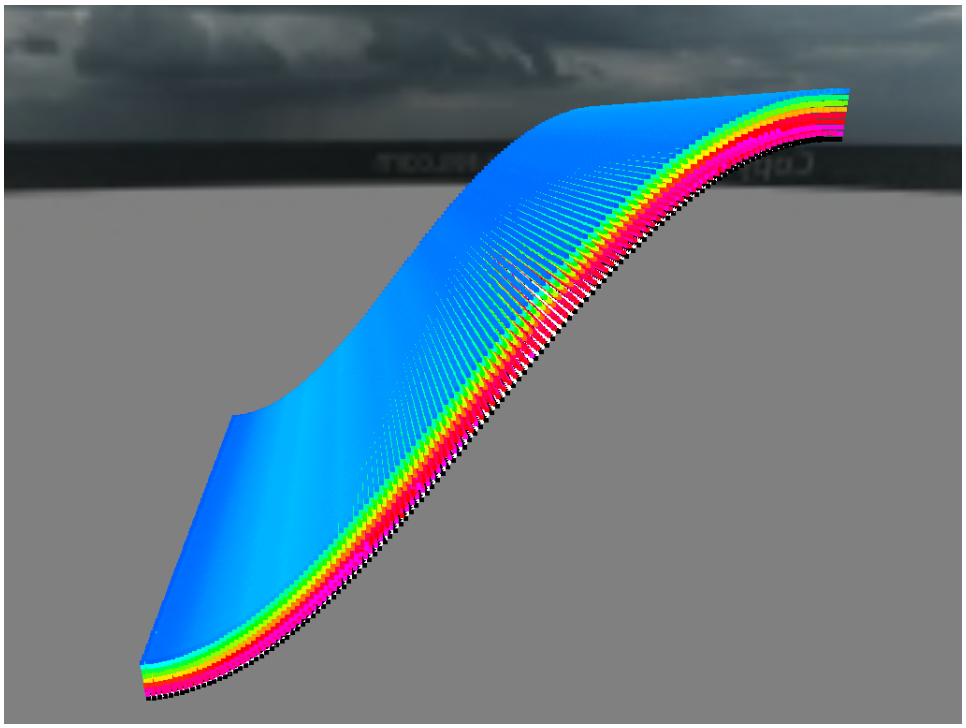


(a) Normal stress

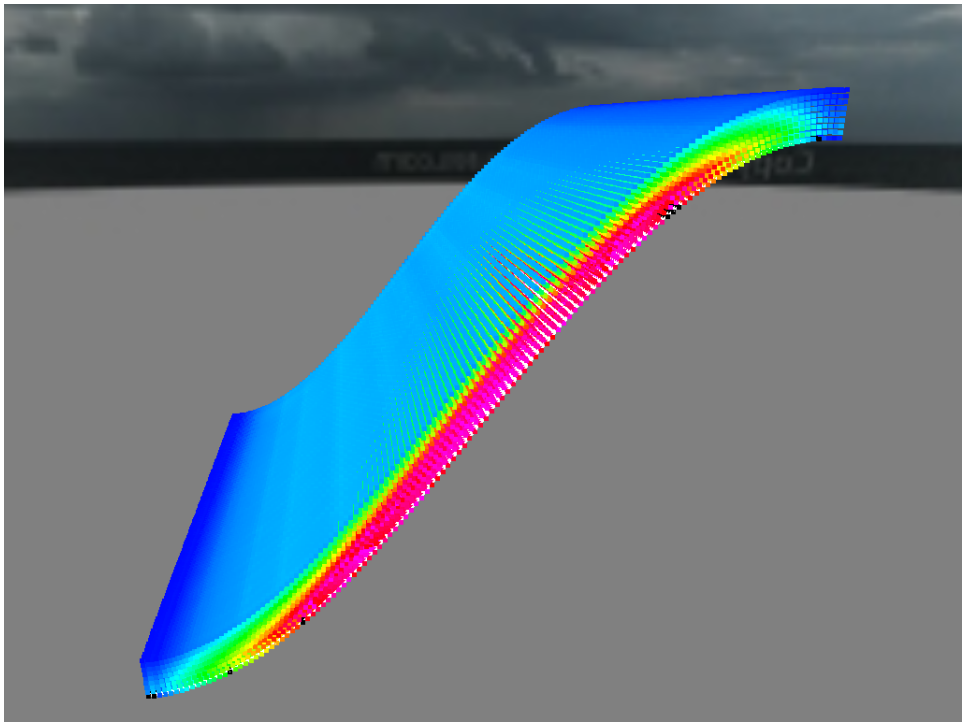


(b) Shear stress

Figure 5.4: Stress distribution for small slope terrain

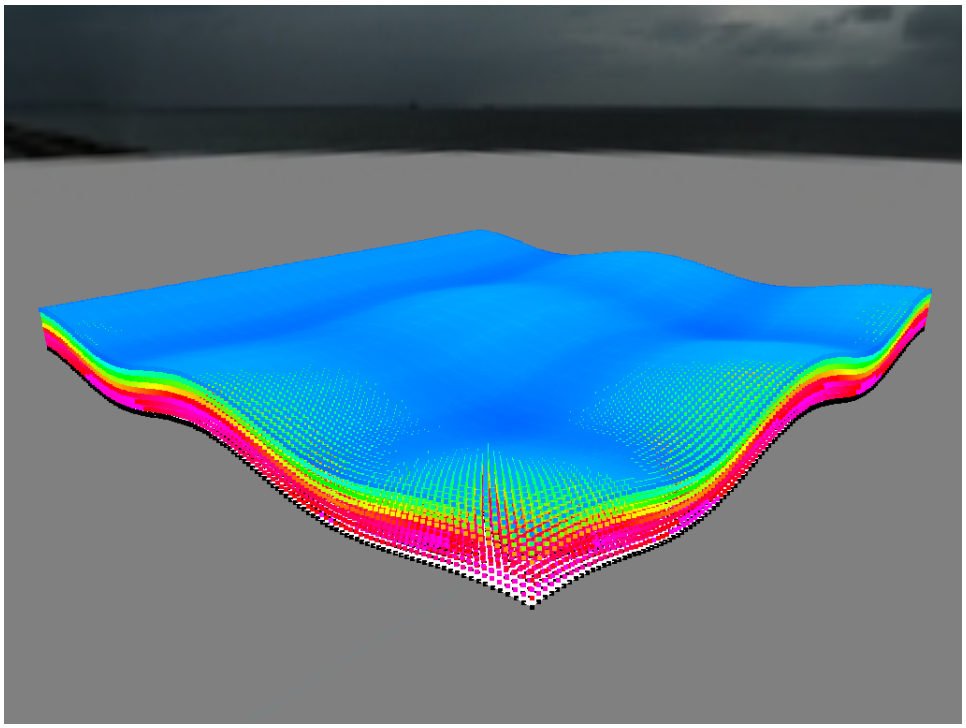


(a) Normal stress

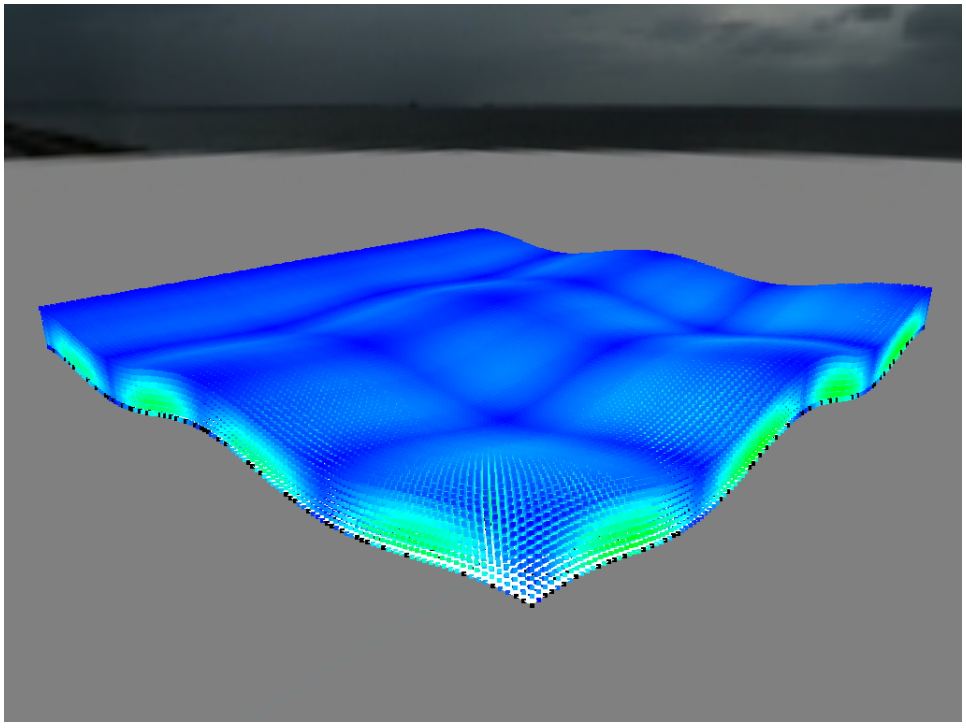


(b) Shear stress

Figure 5.5: Stress distribution for big slope terrain

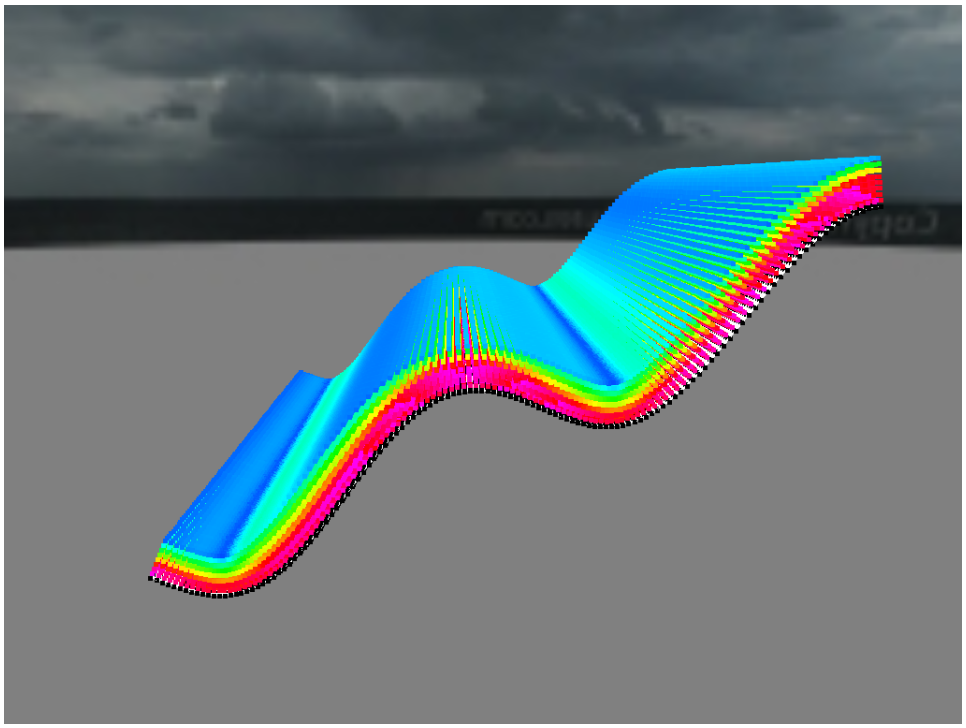


(a) Normal stress

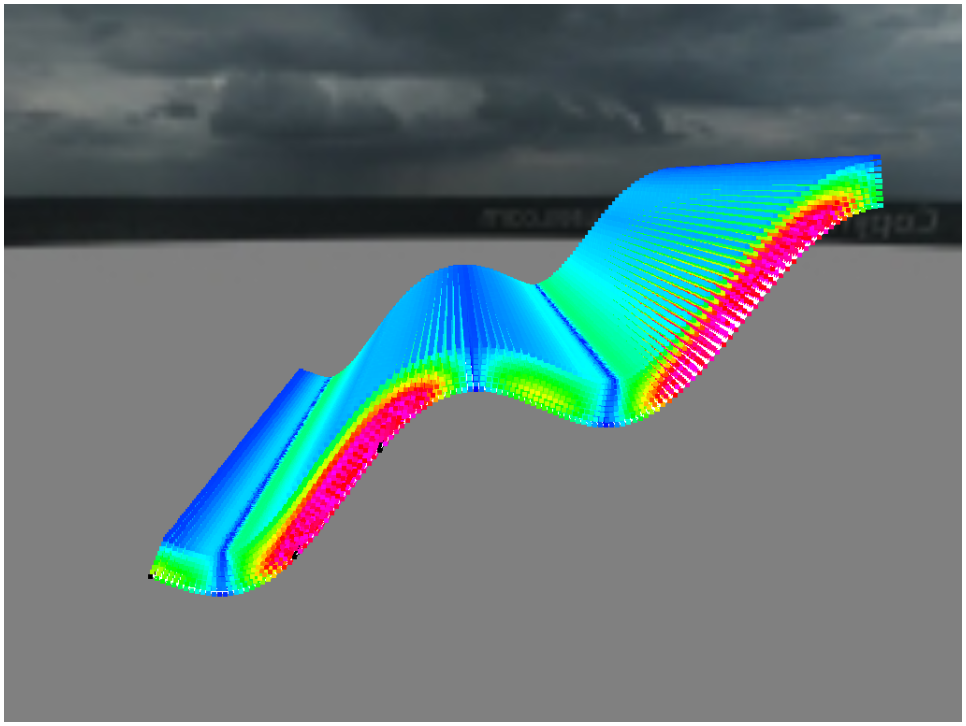


(b) Shear stress

Figure 5.6: Stress distribution for 2D wave terrain



(a) Normal stress



(b) Shear stress

Figure 5.7: Stress distribution for 1D wave terrain

5.2.1.3 Fracture Propagation Distance for Homogeneous Snow

While testing the simulator, it was found that the 'da' parameter, which specifies the length a fracture can propagate when the critical energy release rate is met, has a huge impact on the simulation. And also according to [23], confirmes that the simulation can experience inaccurate results when da is not small enough. However, they do not specify the order of magnitude that this parameter has to be, in order to obtain accurate results. Therefore we will have to test this. In the following tests we use the following parameters:

- Young's modulus $E = 60MPa$
- Critical energy release rate $\mathcal{G}_c = \frac{\left(4.2\times10^{-4}\rho^{2.76}\right)^2}{E}$
- Applied force $\vec{F} = \{0 \;\; 200 \;\; 0\}^T$
- Finite element mass $m = 20kg$
- Spring constant $k = 10000\rho$
- Stress release factor $\gamma = 1000$
- $\omega = 0.1$

The simulation parameters above specifies homogeneous snow (stable), and we should therefore not expect any fractures to occur.

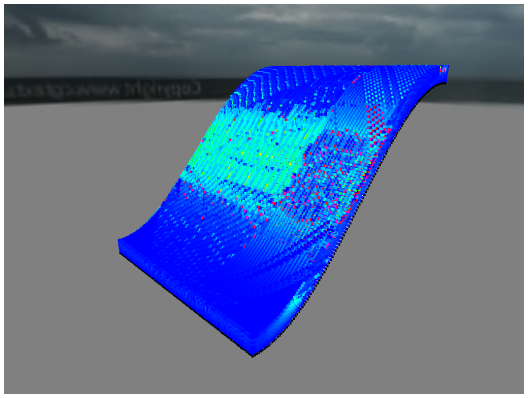
In Figure 5.8 on page 74, the results are shown for testing the simulation impact of the 'da' parameter. And as stated, in the simulation of homogeneous snow, fractures should not occur, and for the results of $da = 10^{-1}$, $da = 10^{-2}$, and $da = 10^{-3}$ displayed in Figures 5.8a, 5.8b, and 5.8c, fractures occurred less than 5 seconds after simulation start.

But when using lower values of $da = 10^{-4}$, $da = 10^{-5}$, $da = 10^{-8}$, and $da = 10^{-11}$, the results are quite similar, and are shown in Figures 5.8d, 5.8e, 5.8f, and 5.8g, and there was not any single fracture occuring. But when using extreme low values of $da = 10^{-12}$, the coloring scheme fails, and starts visualizing purple values, which is shown in Figure 5.8h.

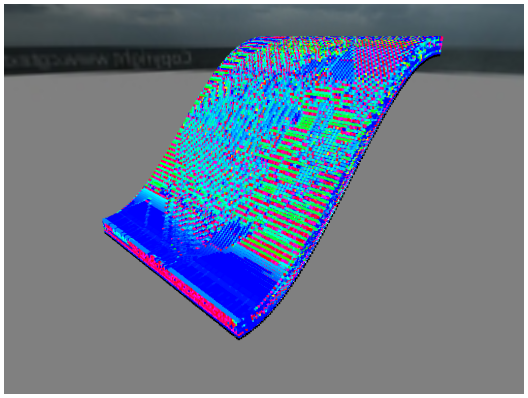
5.2.1.4 Fracture Propagation Distance for Heterogeneous Snow

The 'da' value was also tested for heterogeneous snow (unstable) with the following parameters:

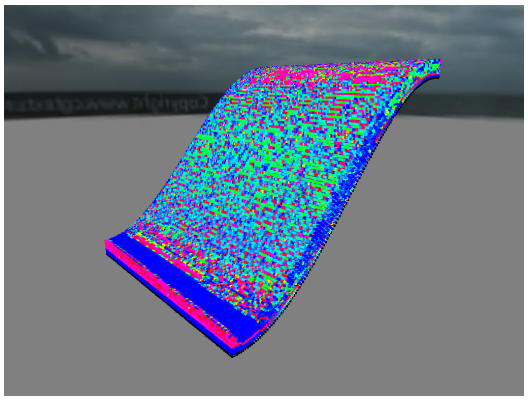
- Young's modulus $E = 1.89\rho^{2.94}Pa$
- Critical energy release rate $\mathcal{G}_c = 0.44J/M^2$
- Applied force $\vec{F} = \{0 \;\; 200 \;\; 0\}^T$
- Finite element mass $m = 20kg$
- Spring constant $k = 10000\rho$
- Stress release factor $\gamma = 1000$
- $\omega = 0.1$



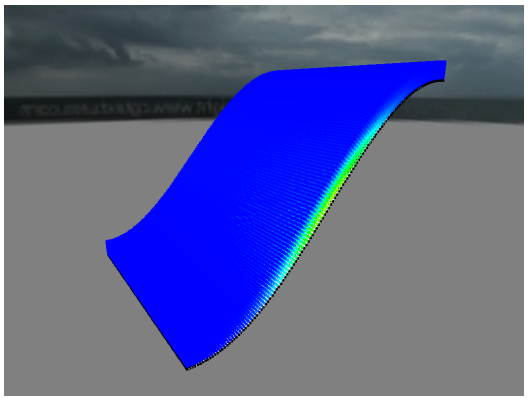
(a) $da = 10^{-1}$



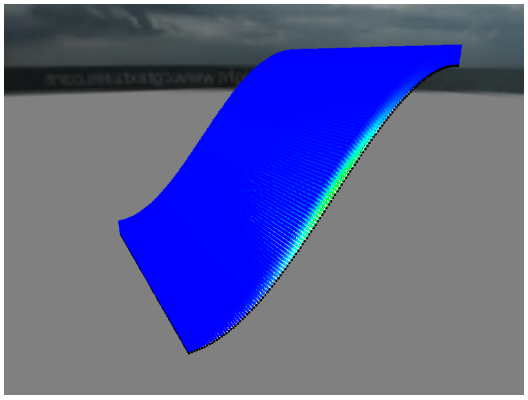
(b) $da = 10^{-2}$



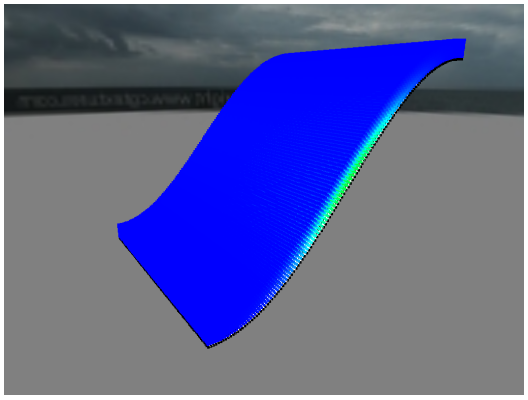
(c) $da = 10^{-3}$



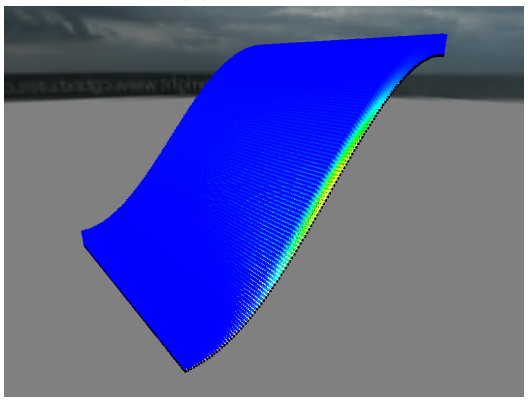
(d) $da = 10^{-4}$



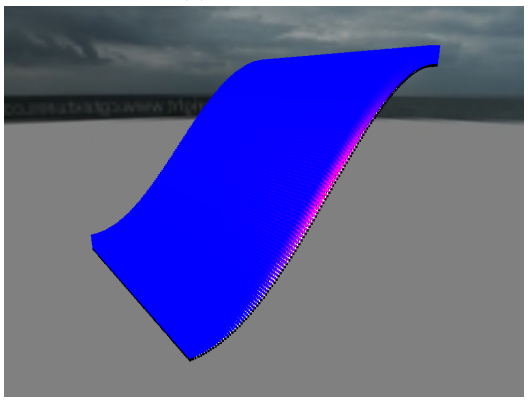
(e) $da = 10^{-5}$



(f) $da = 10^{-8}$

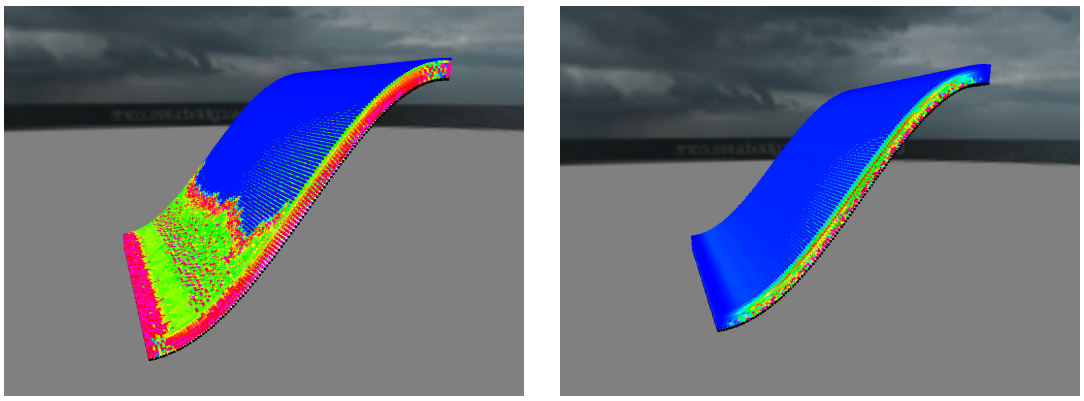


(g) $da = 10^{-11}$



(h) $da = 10^{-12}$

Figure 5.8: Energy ratio for da testing



(a) $da = 10^{-5}$ (b) $da = 10^{-11}$

Figure 5.9: Energy ratio for da testing

And it was then found that the values of $da \geq 10^{-5}$ (which worked for homogeneous snow) generated fractures in all the snow layer, even the upper snow layer. But lower values of da, resulted that fractures was only located in the bottom snow layers, and seemed more correct.

In Figure 5.9a, the fracturing process is shown for using $da = 10^{-5}$, where the red values are indicating that the fractures are extending a distance da. And we can also see that there are fractures occurring in the topmost snow layer, in the bottom of the slope. And these fractures continued to grow toward the top of the slope, covering the whole blue area.

But in Figure 5.9b, there was no fractures occurring in the topmost snow layer, and we ca only see fractures in the bottom snow layers. And since we do not model any movement of the finite elements after failure, I would say that the latter case is a more physically correct modeling of the fracture process, and the process is similar to a fracture propagating in weak snow layers.

5.2.1.5 Energy Release Rate for Homogeneous Snow

Next we will look at the **energy release rate** to the **critical energy release rate** ratio for homogeneous snow. i.e. the energy release rate is normalized to the critical energy release rate, and when this ratio is equal to 1, a fracture process occurs. And we can therefore predict where a fracture will occur before they propagate.

As stated in Section 3.1.3, Christian Sigrist has performed measurement of homogeneous and heterogeneous snow[21]. And his findings of the Young's modulus and the critical energy release rate is found in Table 5.4 on page 76, which will be used for our simulations of homogeneous snow. Below are the parameters used for the following results:

- Young's modulus $E = 60MPa$
- Critical energy release rate $\mathcal{G}_c = \frac{\left(4.2\times10^{-4}\rho^{2.76}\right)^2}{E}$
- Applied force $\vec{F} = \{0 \;\; 200 \;\; 0\}^T$
- Finit element mass $m = 20kg$
- Spring constant $k = 10000\rho$

- Fracture propagation distance $da = 10^{-11}$.
- Stress release factor $\gamma = 1000$
- $\omega = 0.1$

In Table 5.4, we can see that Sigrist found the Young's modulus to exist in a wide range for different types of homogeneous snow. But we will use a value of 60 MPa, in order to generate maximum stress in the snow layers for the highest possibility for fracture propagation, and in the following tests we should not expect any fracture to occur.

Table 5.4: Young's modulus and $\mathcal{G}_c$ for snow

	Heterogeneous snow	Homogeneous snow
$\mathcal{G}_c$	$0.44 \pm 0.020 J/M^2$	$\frac{\left(4.2\times10^{-4}\rho^{2.76}\right)^2}{E}$
Young's Modulus	$1.89\rho^{2.94}Pa$	$[14, 60]$

The results are shown from page 77 to 79, where in Figure 5.10, 5.12, and 5.14 where the terrain is relatively flat, we can see that the energy ratio is near zero across the entire snow structure. However, in Figure 5.11 and 5.13 I got some unexpected results, which may should be expected.

In Figure 5.11, the energy ratio was increasing towards 1 in the steepest locations in the terrains, then a fracture process occurred. However, the snow was then stabilized in the areas where a fracture had propagated. More detailed results are shown in Appendix E.1, where series of screenshots are shown.

Similar results was also obtained when using a very steep slope in Figure 5.13, and in this case, I observed a single fracture propagation in the middle of the slope, which then was followed by a single fracture propagation upward the slope and downward the slope. Then after this single fracture propagation, the snow was then stabilized. Detailed simulation results are shown in Appendix E.2.

Also when simulating a more complex terrain consisting of two slopes in Figure 5.15, there occurred some fractures. But in this terrain, there was more than one single fracture, and after approximately 1 minute of simulation time with an average FPS of 26, the snow was stabilized.

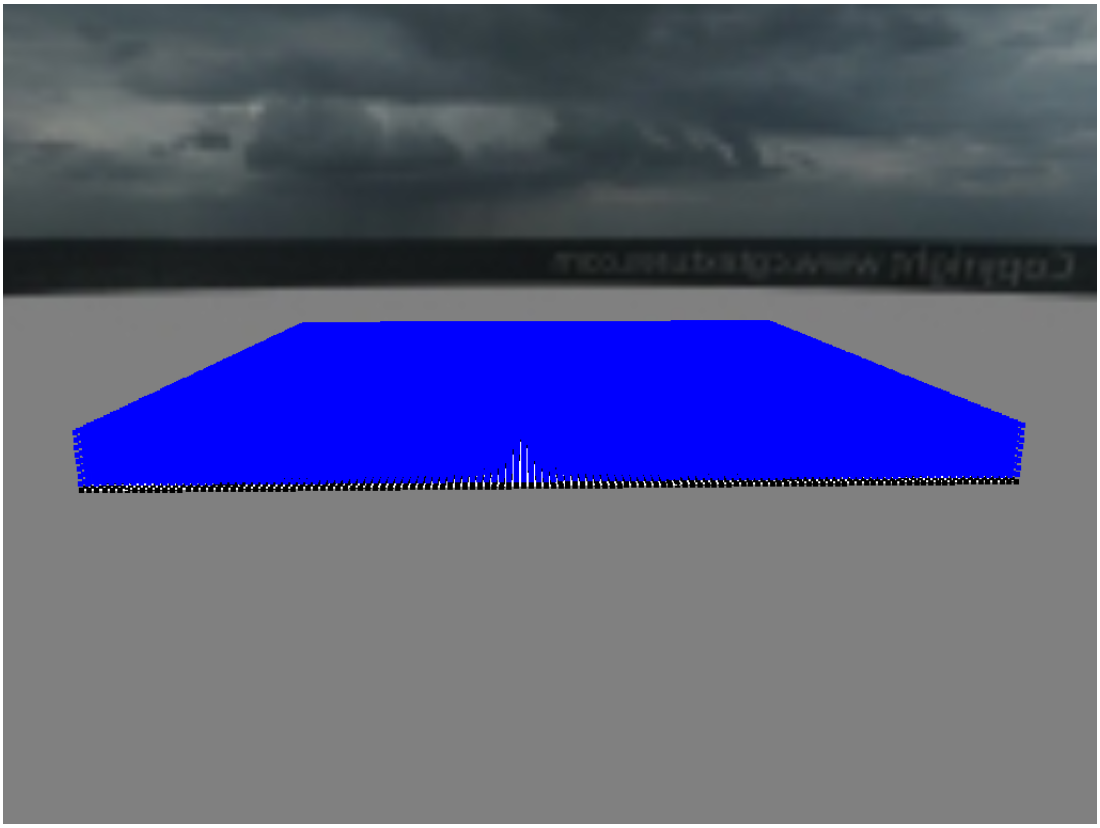


Figure 5.10: Energy ratio for flat terrain, homogeneous snow

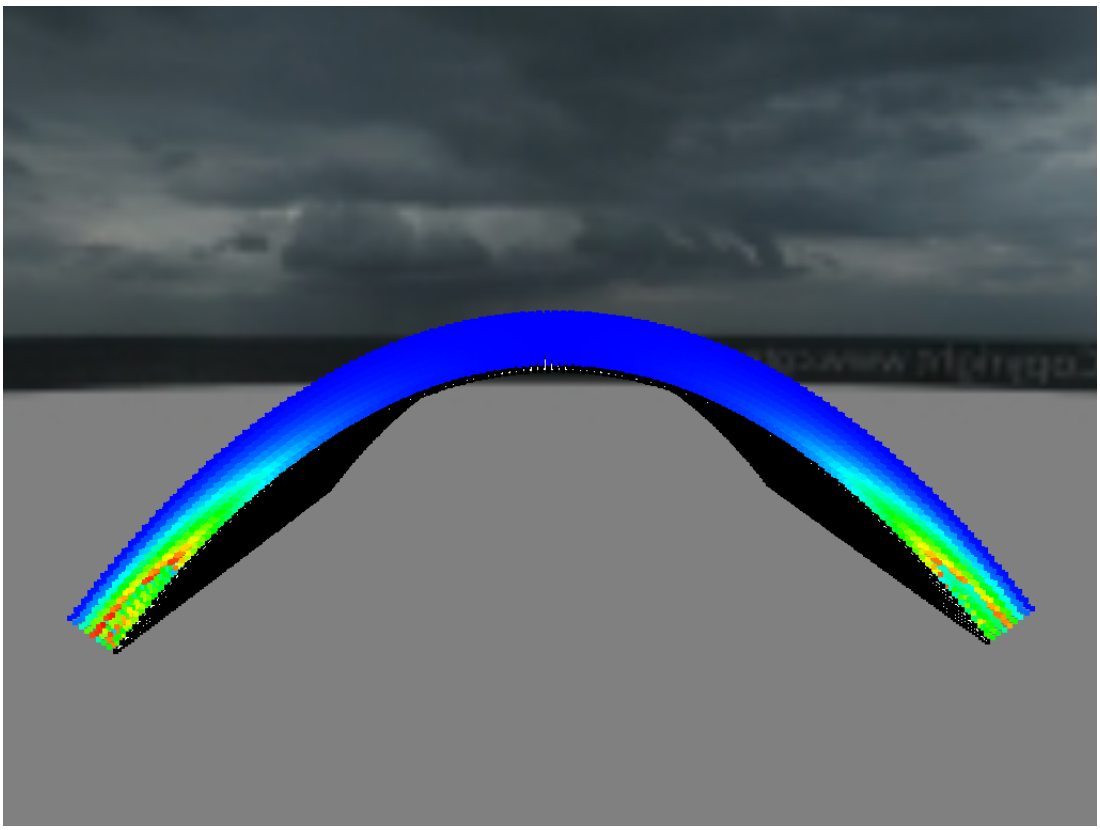


Figure 5.11: Energy ratio for parabola terrain, homogeneous snow

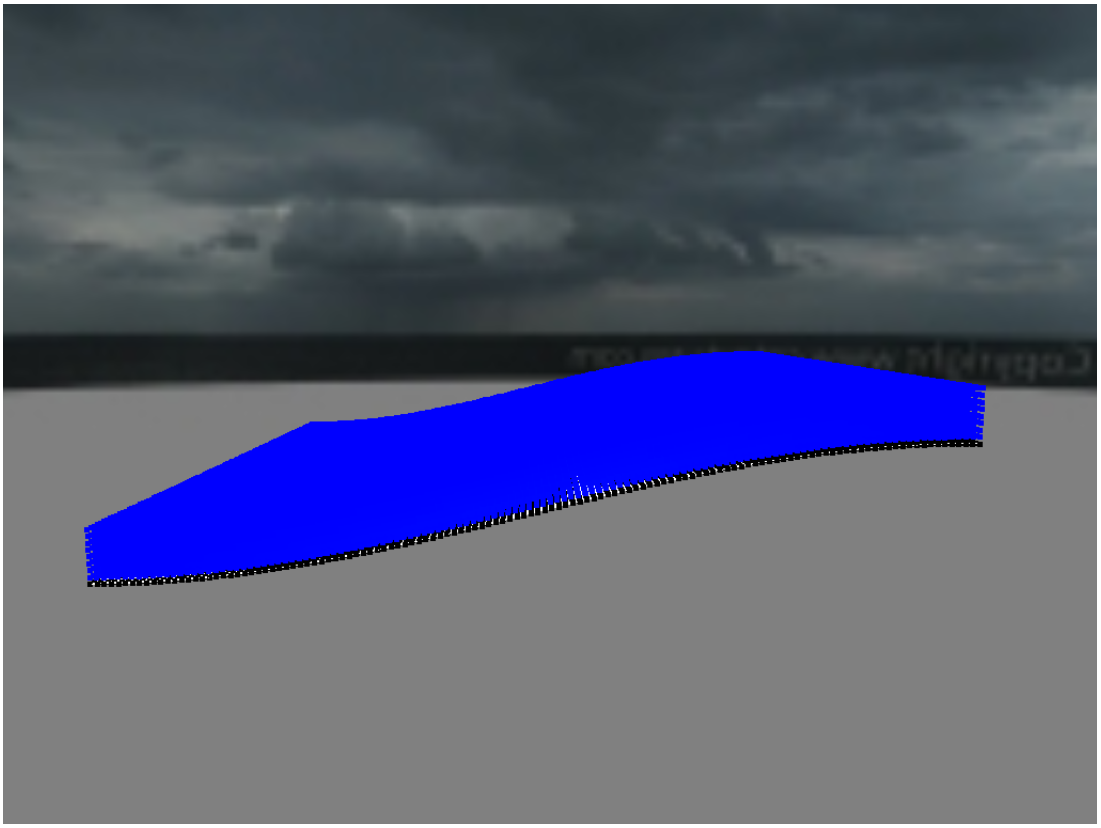


Figure 5.12: Energy ratio for small slope terrain, homogeneous snow

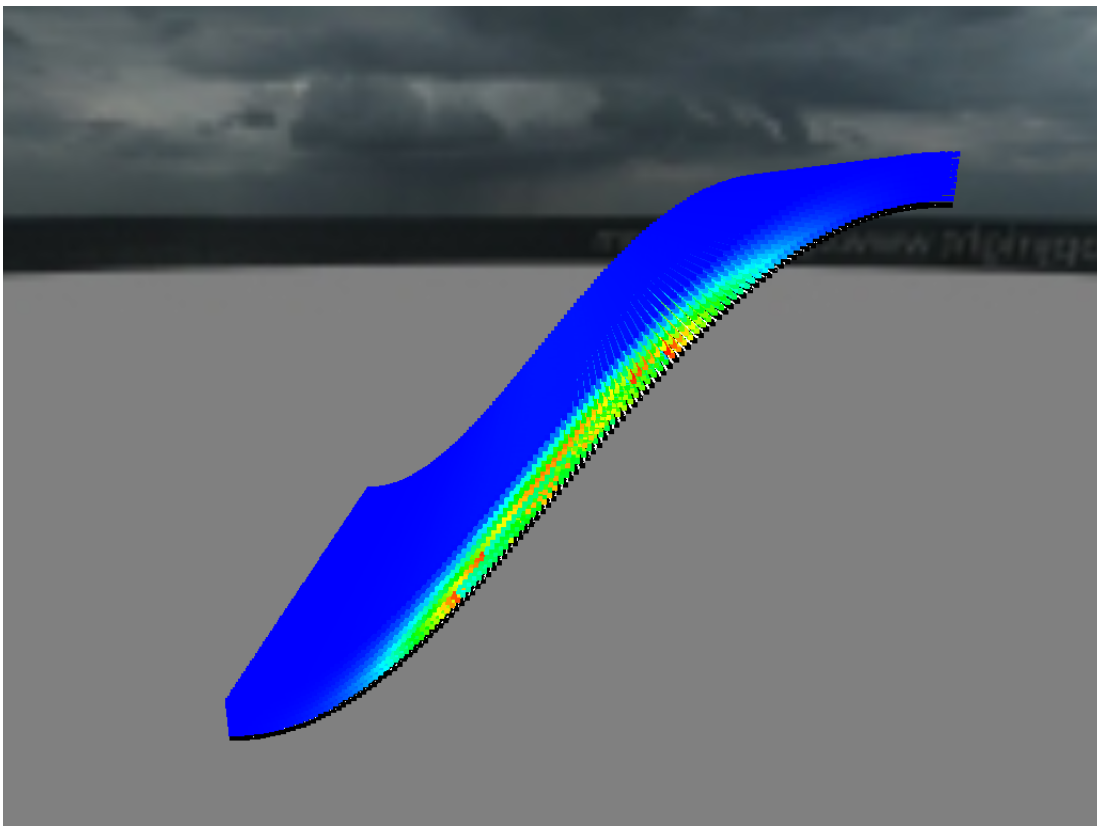


Figure 5.13: Energy ratio for big slope terrain, homogeneous snow

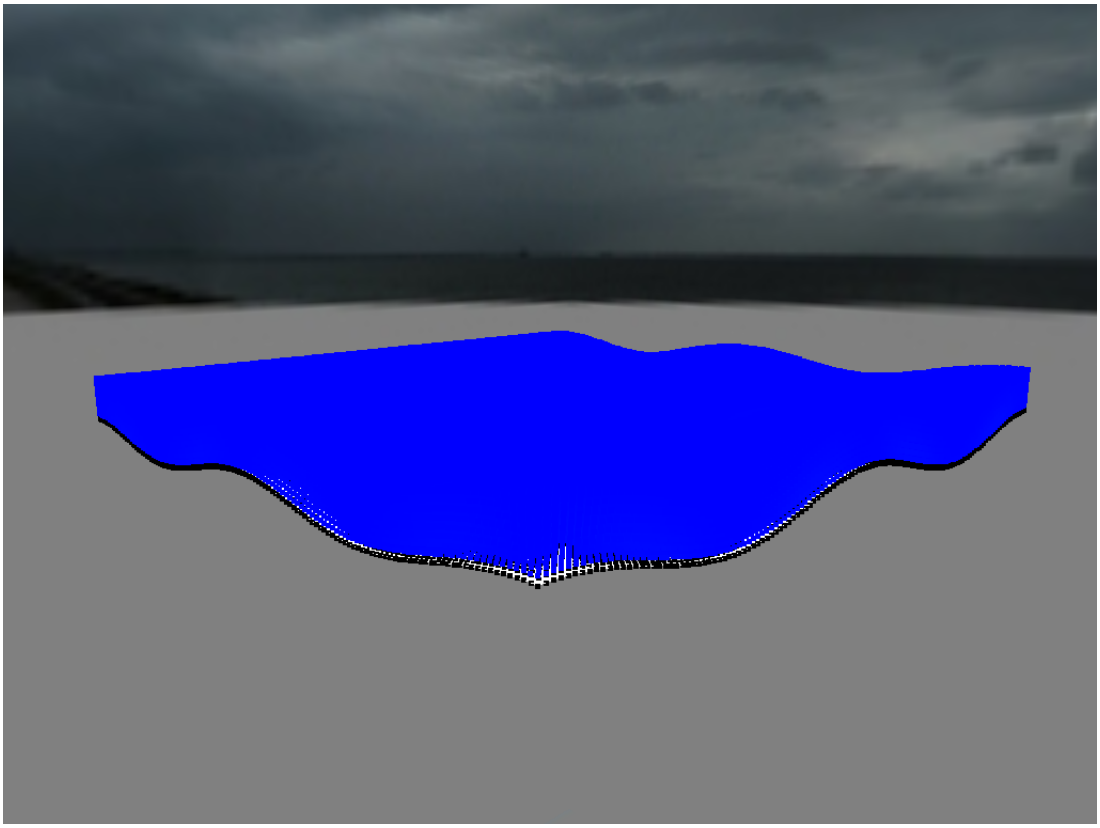


Figure 5.14: Energy ratio for 2D wave terrain, homogeneous snow

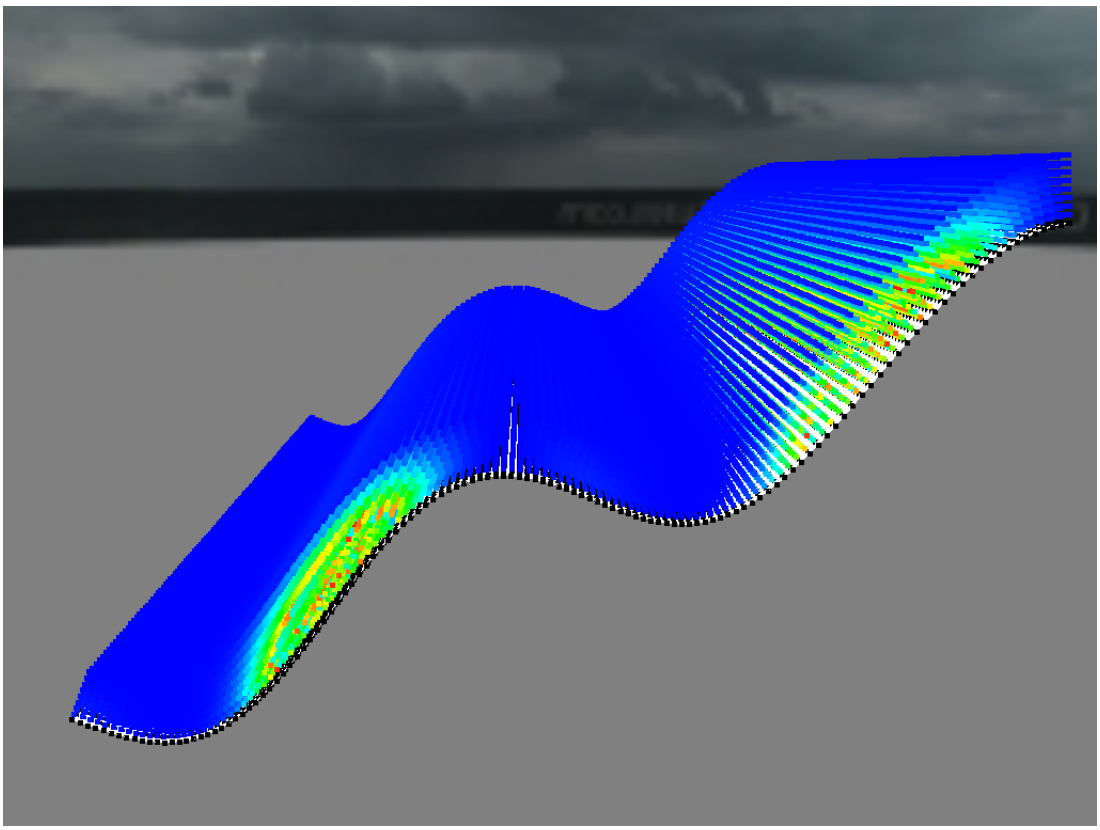


Figure 5.15: Energy ratio for 1D wave terrain, homogeneous snow

5.2.1.6 Energy Release Rate for Heterogeneous Snow

In these tests, we will simulate heterogeneous snow. And as specified in Table 5.4 on page 76, the simulation parameters will therefore be the following:

- Young's modulus $1.89\rho^{2.94}Pa$
- Critical energy release rate $0.44J/M^2$
- Applied force $\vec{F} = \{0 \;\; 200 \;\; 0\}^T$
- Finite element mass $m = 20kg$
- Spring constant $k = 10000\rho$
- Fracture propagation distance $da = 10^{-11}$.
- Stress release factor $\gamma = 1000$
- $\omega = 0.1$

The results are shown from page 81 to 83, and first in Figure 5.16 where the results from the flat terrain is shown, we do not observe any energy available for a fracture process to occur.

But in Figure 5.17, where a parabola terrain was simulated, we could observe the energy ratio increasing up towards a fracture propagation, then afterwards, fractures was continuously propagating. The same result was also obtained for the steep slope terrain in Figure 5.19 and 5.21. And in all cases, major parts of the terrain was experiencing fractures.

Then lastly in Figure 5.18 and 5.20, where the terrain simulated was more flat, we could also observe fractures propagating. But in both cases, it was only in relative small parts of the terrain.

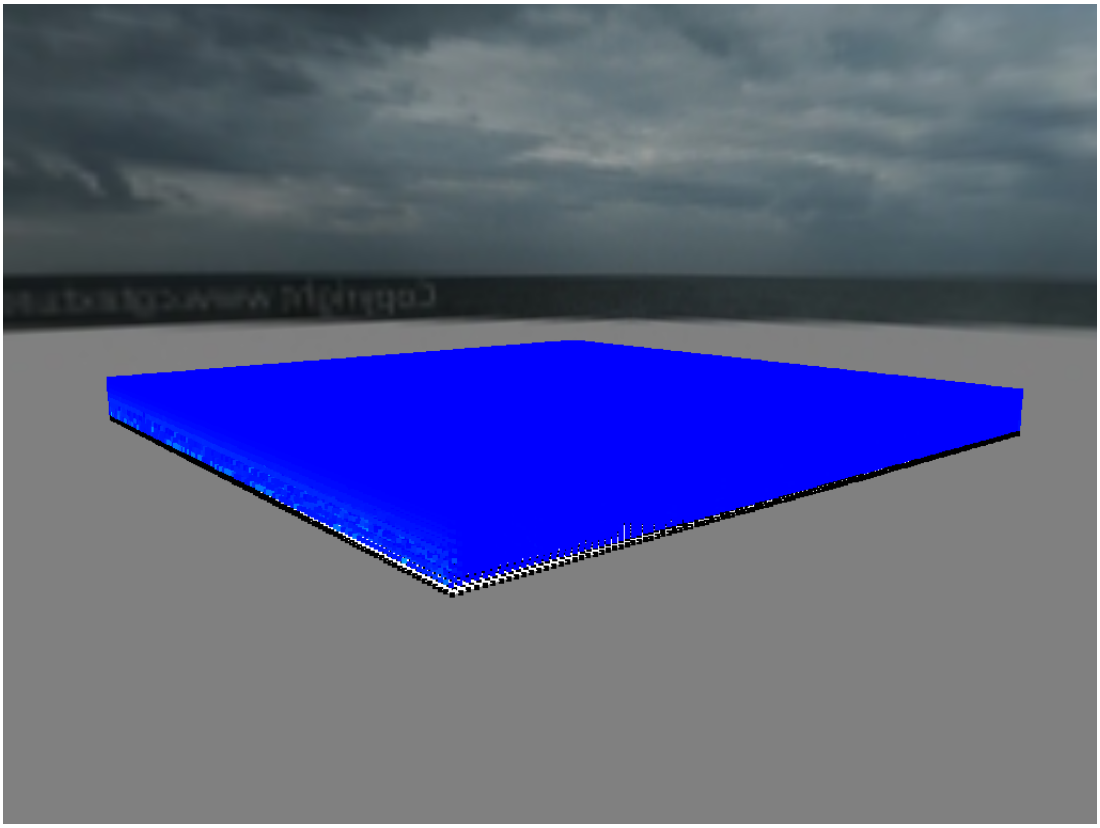


Figure 5.16: Energy ratio for flat terrain, heterogeneous snow

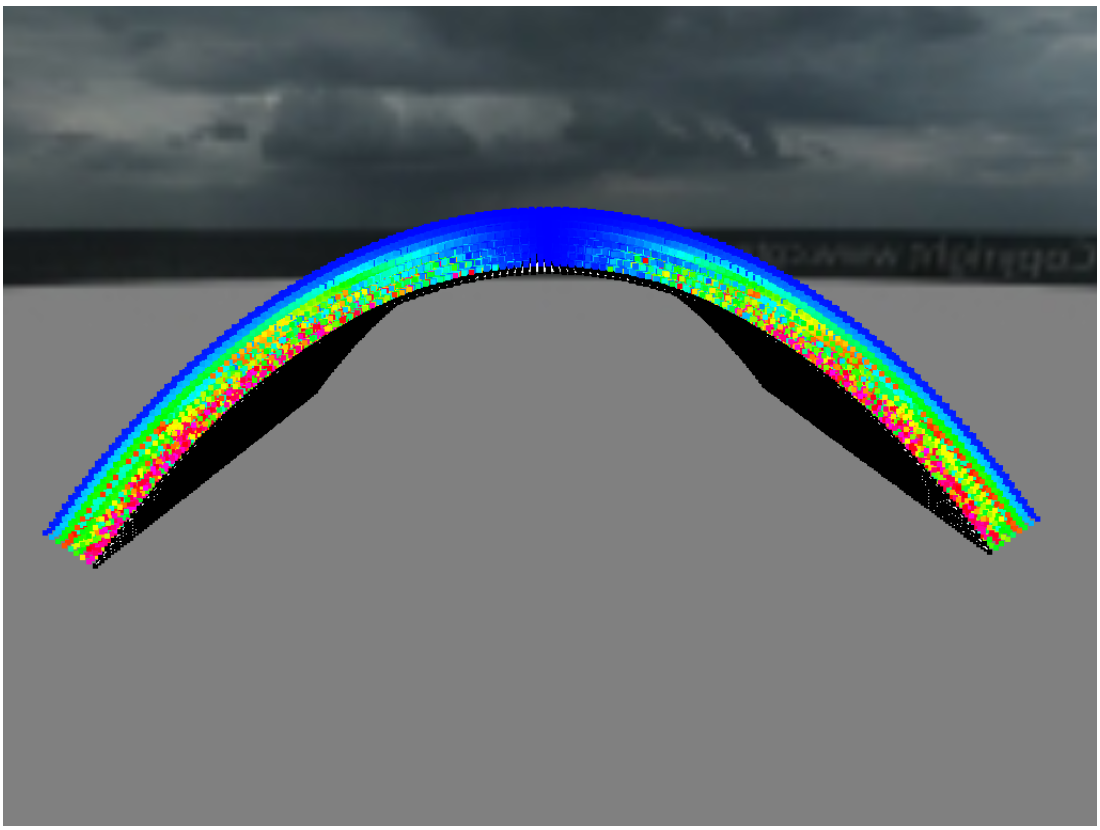


Figure 5.17: Energy ratio for parabola terrain, heterogeneous snow

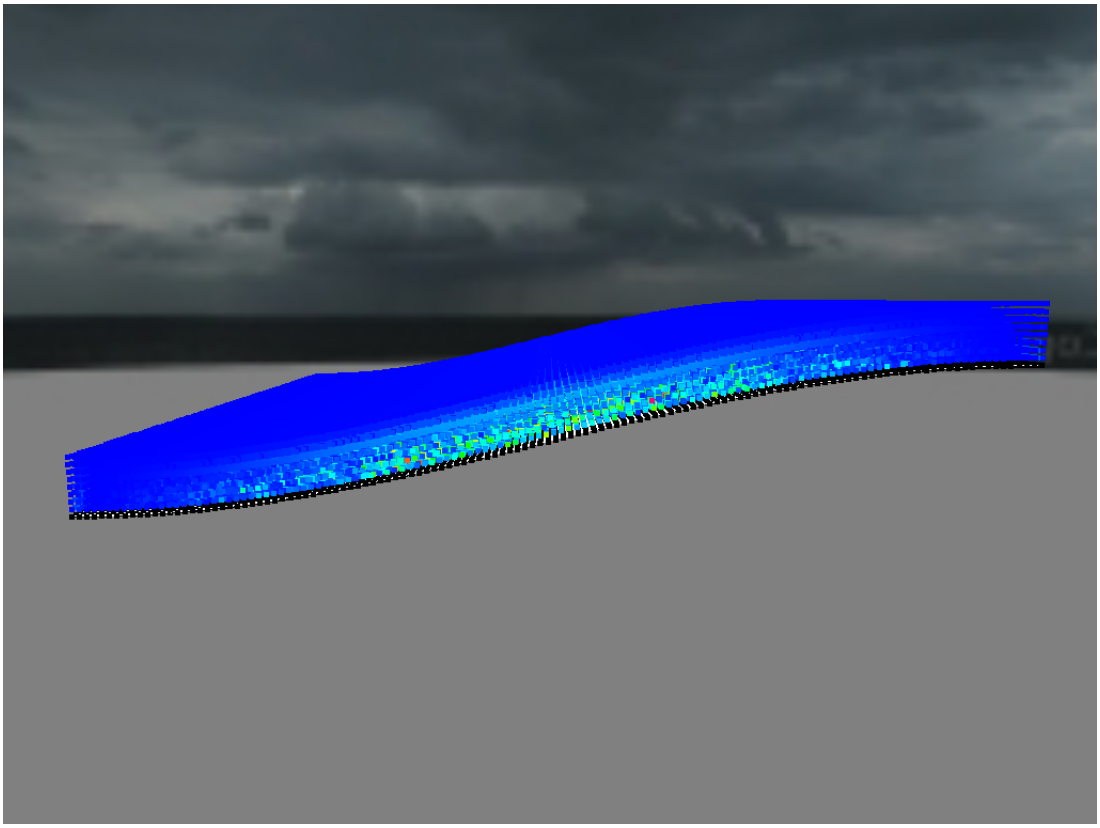


Figure 5.18: Energy ratio for small slope terrain, heterogeneous snow

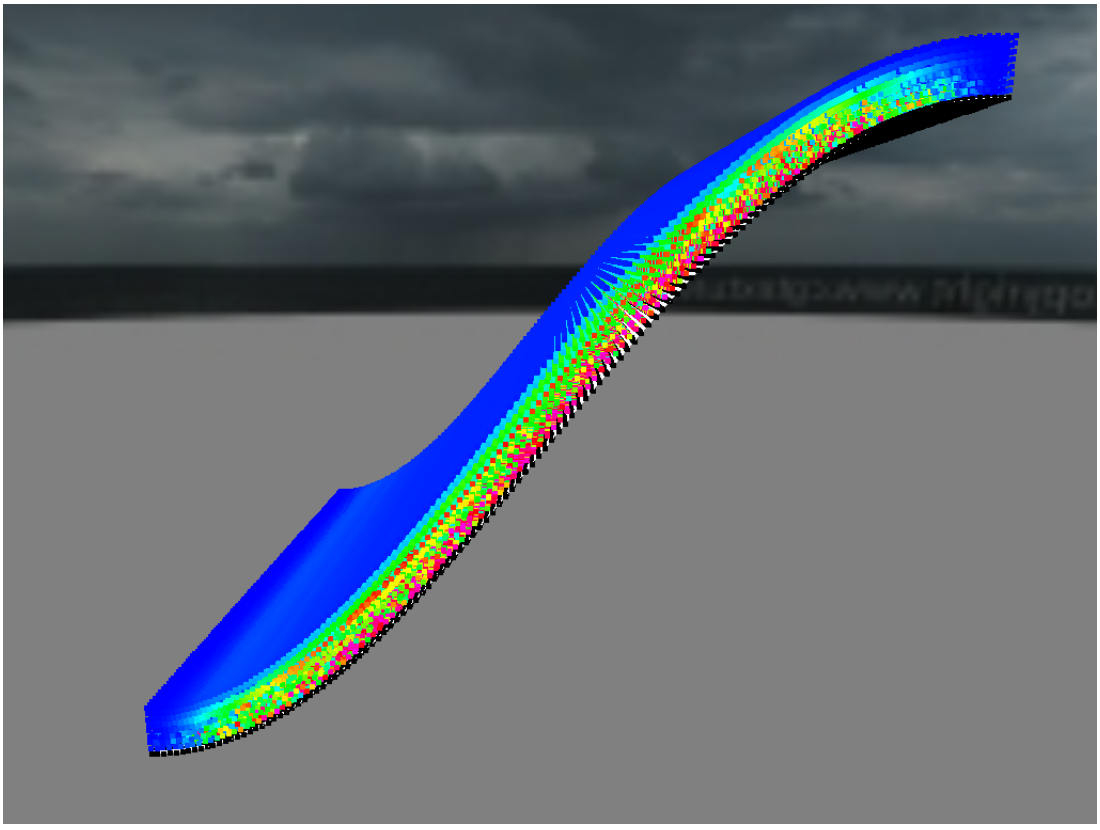


Figure 5.19: Energy ratio for big slope terrain, heterogeneous snow

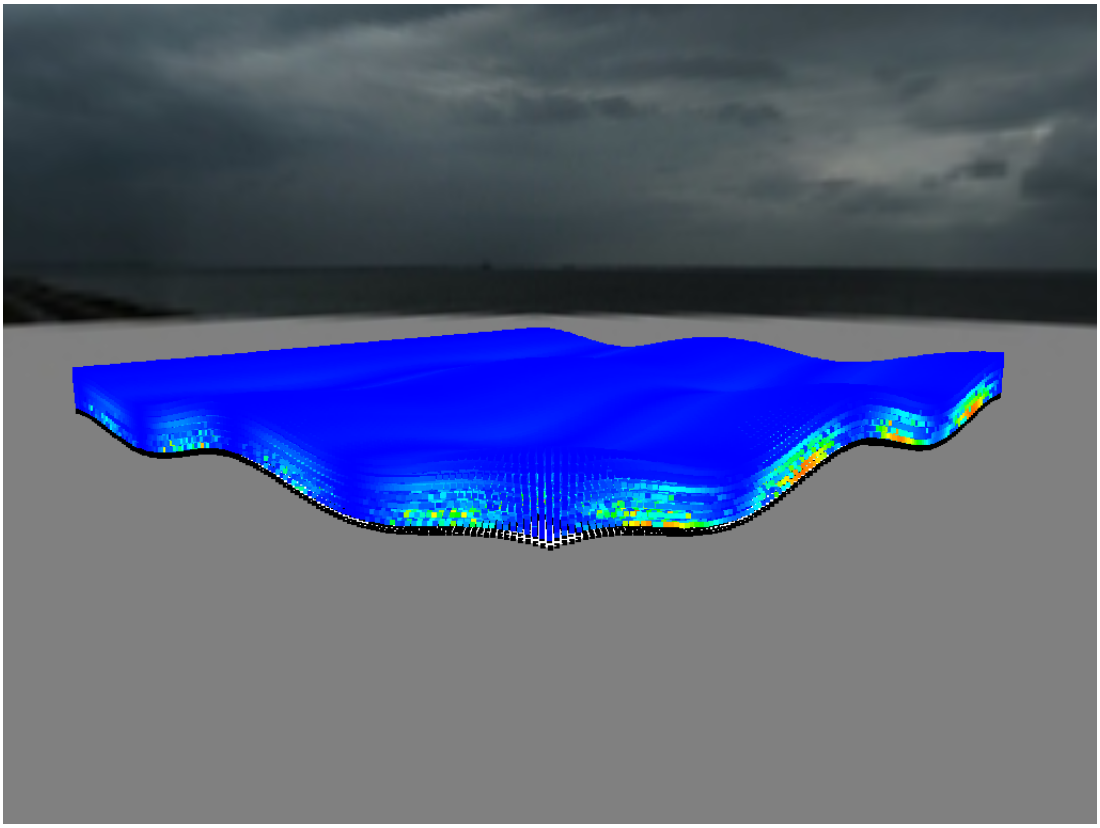


Figure 5.20: Energy ratio for 2D wave terrain, heterogeneous snow

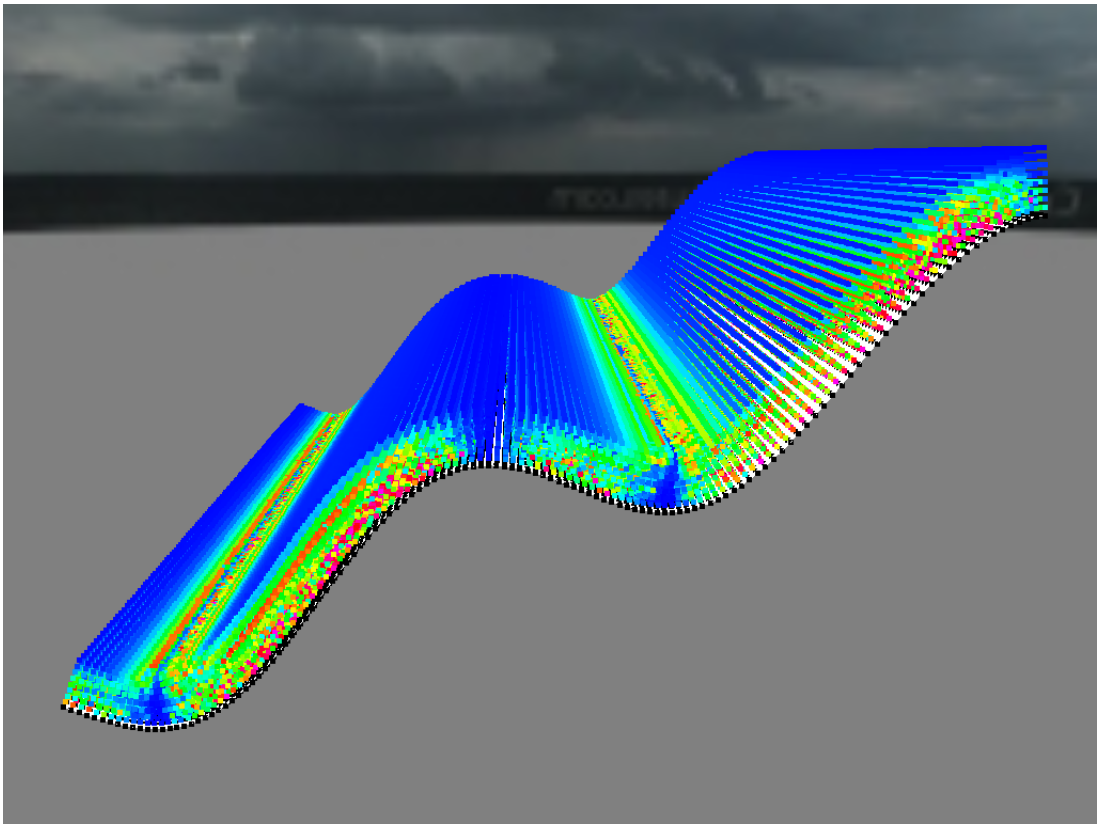


Figure 5.21: Energy ratio for 1D wave terrain, heterogeneous snow

5.2.2 Performance Results

In the following sections, we will look into different performance aspects of the simulator. And I will also emphasize that although both workstation 1 and 2 has multiple GPUs, the calculations are only being performed on the Tesla GPU present in both workstations. The GeForce GPU is only used as a rendering device, and are not being heavily utilized.

However, there is an exception in Section 5.2.2.3, where we look into the performance penalty of using double precision when calculating the stress, where we separately tests all 4 GPUs in the two workstations. This is due to that Tesla GPUs has a ratio of 1:2 between single and double floating points units, and the GeForce GPUs only have a ratio of 1:24. And therefore a GeForce GPU should have a higher penalty of using double precision when calculating the stresses in the snow.

5.2.2.1 Kernel Launch Configuration Analysis

In this section, we will look into the performance of the different kernels in the simulator by varying number of threads per block, where both the grid and blocks are specified in three dimensions. In this project, there are a total of 16 kernels, but 8 of them is used to calculated the displacement of the exterior nodes of each finite element, and they differ only in array indexing. And the other 8, is used to calculate the energy release rate for each exterior node of the finite elements, and these also differ only in array indexing. Therefore I will groupe these kernels into the following:

- *solve_global_displacement*
- *propagate_fractures*

And when properties about these kernels are discussed, e.g execution time, it will always be the sum of the 8 kernels that are listed in this report.

First we will try different launch configuration when using the Tesla C2070. The results are shown in Table 5.5 on page 85, when running the simulator for 100 frames. And from this we can see that the ideal launch configuration when using the Tesla C2070 is $\{128, 1, 4\}$, which sums up to a total of 512 threads. And this is also the maximum number of thread per block allowed on the Fermi architecture.

Next, the same tests was performed on the Tesla K40c which has a Kepler architecture, and the results are shown in Table 5.6 on page 85. And for the tests performed on the Kepler architecture, we could increase the total number of threads per block to 1024 for the *solve_global_displacement* kernel. And the best combination of thread per block was found to be $\{128, 1, 8\}$. However, the *propagate_fractures* kernel could not utilize 1024 thread. This was caused by the total number of registers per thread used in these kernels, which limits the number of threads per block, because each block has a finite number of registers available. Therefore the maximum number of threads we could use for the *propagate_fractures* kernel was 512, and the best combination was $\{128, 1, 4\}$.

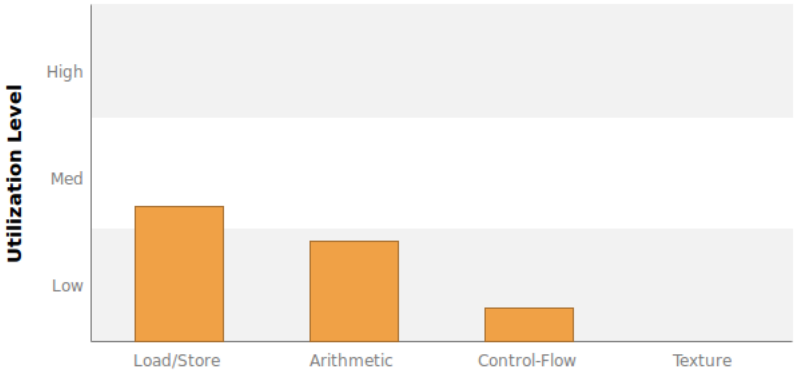
Also note that both tables shows the average execution time for both kernels for a single time step.

Table 5.5: kernel time varying threads per block, Tesla C2070

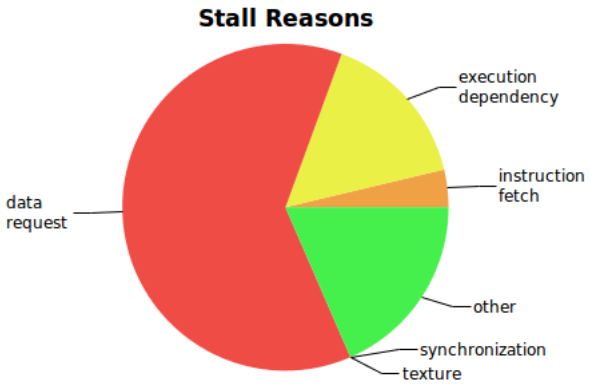
Config	Thread per block	solve global displacement Avg time	propagate fractures Avg time
$\{2, 2, 2\}$	8	6.814 ms	42.2360 ms
$\{3, 3, 3\}$	27	3.799 ms	17.5919 ms
$\{4, 4, 4\}$	64	2.957 ms	10.3488 ms
$\{5, 5, 5\}$	125	2.798 ms	11.6998 ms
$\{6, 6, 6\}$	216	2.597 ms	13.6946 ms
$\{7, 7, 7\}$	343	2.649 ms	17.2157 ms
$\{8, 8, 8\}$	512	2.523 ms	10.0027 ms
$\{16, 2, 16\}$	512	1.658 ms	8.7159 ms
$\{32, 1, 16\}$	512	1.459 ms	8.3857 ms
$\{64, 1, 8\}$	512	1.434 ms	8.3128 ms
$\{128, 1, 4\}$	512	**1.427 ms**	**8.2756 ms**
$\{256, 1, 2\}$	512	1.877 ms	10.9222 ms
$\{512, 1, 1\}$	512	3.155 ms	18.5747 ms

Table 5.6: kernel time varying threads per block, Tesla K40c

Config	Thread per block	solve global displacement Avg time	propagate fractures Avg time
$\{2, 2, 2\}$	8	5.99169 ms	19.4722 ms
$\{3, 3, 3\}$	27	4.33712 ms	10.8056 ms
$\{4, 4, 4\}$	64	3.51982 ms	8.6887 ms
$\{5, 5, 5\}$	125	2.35444 ms	8.7245 ms
$\{6, 6, 6\}$	216	1.76514 ms	10.9877 ms
$\{7, 7, 7\}$	343	1.38036 ms	15.0574 ms
$\{8, 8, 8\}$	512	0.96504 ms	8.75090 ms
$\{16, 2, 16\}$	512	——	7.76228 ms
$\{32, 1, 16\}$	512	——	7.68959 ms
$\{64, 1, 8\}$	512	——	7.70778 ms
$\{128, 1, 4\}$	512	——	**7.58560 ms**
$\{256, 1, 2\}$	512	——	10.1087 ms
$\{9, 9, 9\}$	729	0.63258 ms	failure
$\{10, 10, 10\}$	1000	0.36331 ms	failure
$\{16, 4, 16\}$	1024	0.35049 ms	failure
$\{32, 2, 16\}$	1024	0.26429 ms	failure
$\{64, 1, 16\}$	1024	0.10897 ms	failure
$\{128, 1, 8\}$	1024	**0.10877 ms**	failure
$\{256, 1, 4\}$	1024	0.19915 ms	failure



(a) Intruction utilization



(b) Instruction stall reasons

Figure 5.22: Tesla C2070 Utilization, solve_global_displacement_kernel_step1

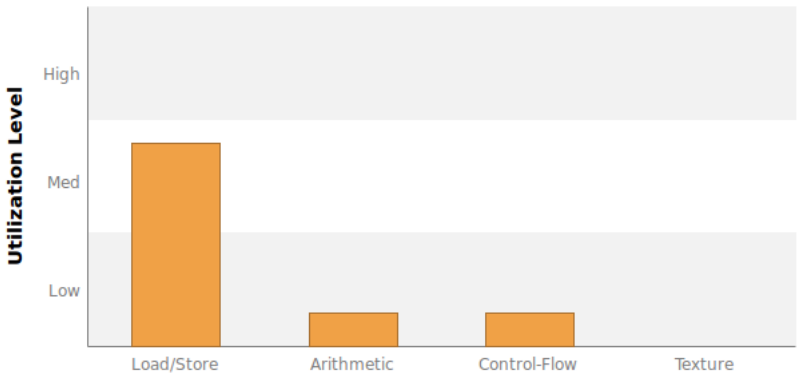
5.2.2.2 Kernel Analysis

In this section, we will look into the GPU utilization of both kernels, and we will use the *Nvidia visual profiler* to obtain our results. But since this tool only allows us to collect data about one kernel simultaneously, we will only gather data for one of the eight kernels for both *solve_global_displacement* and *propagate_fractures*, namly *solve_global_displacement_step1* and *propagate_fractures_step1*. The tests will also be performed on the Fermi and Kepler architecture using the Tesla C2070 and K40c.

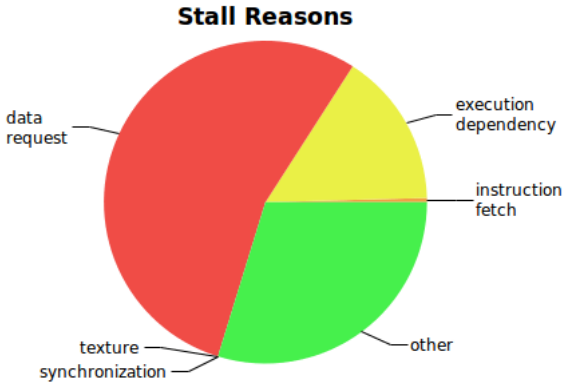
solve_global_displacement_step1: Performance test of the different instruction utilization was first performed with the Tesla C2070, and as Figure 5.22a shows, the utilization of different kind of instructions are not any performance limiter in our simulation. Further in Figure 5.22b we can examine the instruction stall reason, and as suspected, the *data request* is the major part of the instruction stalls. This is due to vast amount of memory reads that each thread performs.

The same tests was also performed on the Tesla K40c, and in Figure 5.23a on page 87 we can see the instruction utilization, and in Figure 5.23b on page 87 we can see the instruction stall reason. And for the Kepler architecture, we can see that the Load/Store instructions are experiencing a higher utilization than in the Fermi architecture, and the arithmetic instructions are about 33% than in the Fermi. But again, for these test, the instruction utilization is not any performance limiter. It is due to the amount of memory that each thread needs.

In Table 5.7 on page 88, the memory bandwidth for both test GPUs are shown.



(a) Intruction utilization



(b) Instruction stall reasons

Figure 5.23: Tesla K40c Utilization, solve_global_displacement_step1

The *"Utilization"* column uses a scale that goes from 0-10, where 0 indicates a "Idle" state, 1 equals low utilization, 5 is medium, 8 is high, and 10 is the device maximum. A discussion with Eirik Myklebost (which performed detailed hardware analysis of Nvidia GPU architecture in his specialization project[15]), gave some additional information about the memory utilization of Fermi versus Kepler.

The Fermi architecture automatically uses the fast on-chip L1 cache for all memory operations, and therefore we obtain a relative high L1 cache utilization on the Fermi architecture with a value of 6. But if we compare to the Kepler architecture, we only obtain a utilization of 2. This is because in the Kepler architecture, the GPU will only use the fast L1 cache for a thread's local memory, and if there are enough registers to avoid any local memory usage, the L1-cache will not be used.

We can also see in Table 5.7, that the Kepler architecture has a total L1 cache bandwidth usage of 301.11 GB/s. But if this is compared to the total L2 cache bandwidth usage of 301.18 GB/s, which differs only in 0.07 GB/s, we can see that all L1 reads are resulting in a cache-miss, and the GPU must search in the L2 cache where the data is located. i.e. The L1 cache is not used at all in the Kepler architecture.

Table 5.7: Memory bandwidth, solve global displacement kernel

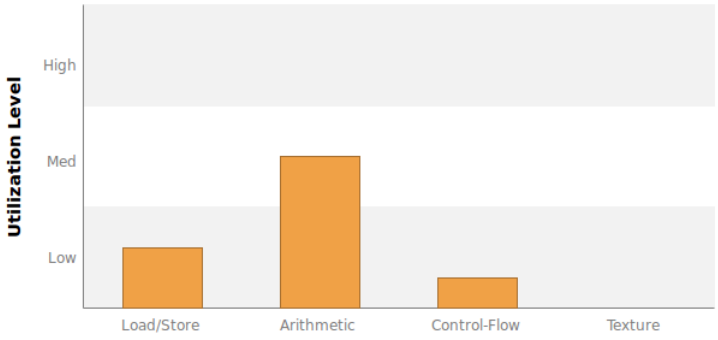
	Tesla C2070		Tesla K40c	
L1 / Shared	Bandwidth	Utilization	Bandwidth	Utilization
Local loads	0.00 GB/s		0.00 GB/s	
Local stores	0.00 GB/s		0.00 GB/s	
Shared loads	0.00 GB/s		0.00 GB/s	
Shared stores	0.00 GB/s		0.00 GB/s	
Global loads	473.72 GB/s		286.67 GB/s	
Global stores	21.95 GB/s		14.44 GB/s	
Total	495.67 GB/s	6	301.11 GB/s	2
L2 Cache				
Reads	49.27 GB/s		286.74 GB/s	
Stores	21.95 GB/s		14.44 GB/s	
Total	71.23 GB/s	4	301.18 GB/s	6
Device Memory				
Reads	26.27 GB/s		21.69 GB/s	
Stores	14.28 GB/s		10.47 GB/s	
Total	40.55 GB/s	4	32.16 GB/s	2

propagate_fractures_step1: Instruction utilization, instruction stalls, and memory utilization tests was also performed on one of the eight kernels which calculates the energy available for fracture propagation. And the tests was performed with a GPU with Fermi and Kepler architecture.

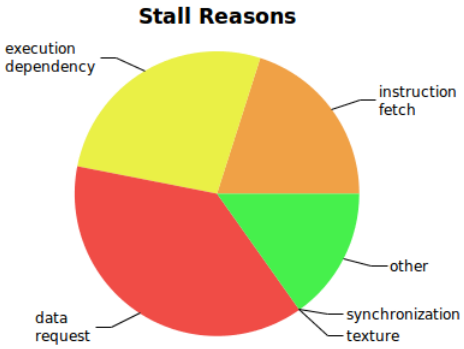
First we can see the instruction utilization in Figure 5.24a on page 89 when using the Tesla C2070 (Fermi), and since this kernels are far more complex in arithmetics than the previous kernel discussed, we can see this reflect in the utilization of the arithmetic hardware units. But none of the instruction units are utilized so much that they become any performance limiter in this case either. And in Figure 5.24b on page 89 we can see the instruction stall reasons, where we can see that the *"data request"* portion is relatively lower compared to the previous kernels, and the *"execution dependency"* and *"instruction fetch"* are becoming larger.

Next up, we performed the same tests on the Tesla K40c (Kepler), and we can see the same tendencies as the Fermi. The Load/Store unit is being less utilized, and the arithmetic unit is about 2x larger (Figure 5.25a on page 89), however none of them is becoming any performance limiter. And in Figure 5.25b on page 89, we can see that the *"data request"* stall reason is becoming smaller than the previous kernel, and the *"execution dependency"* and *"instruction fetch"* are increasing.

Next we will look into the memory utilization for both architectures, and the results are shown in Table 5.8 on page 90, and compared to the previous kernel, the overall memory usage is relative lower in all *'total'* fields. But in this case, we can see that in the L1/Shared memory section, the Tesla C2070 are now using the thread's local memory. Which is caused by the complexity of the kernel, and the threads are using more registers than available. And again, we can see that the Kepler GPU does not use any of the L1 cache.

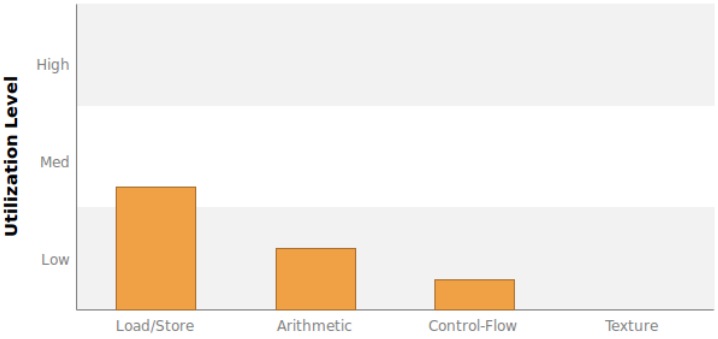


(a) Instruction utilization

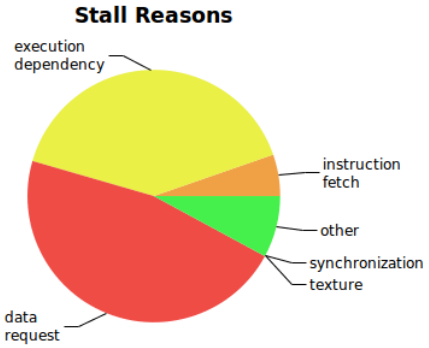


(b) Instruction stall reasons

Figure 5.24: Tesla C2070 Utilization, propagate_fractures_step1



(a) Instruction utilization



(b) Instruction stall reasons

Figure 5.25: Tesla K40c Utilization, propagate_fractures_step1

Table 5.8: Memory bandwidth, propagate fracture kernel

	Tesla C2070		Tesla K40c	
L1 / Shared	Bandwidth	Utilization	Bandwidth	Utilization
Local loads	25.64 GB/s		0.00 GB/s	
Local stores	21.21 GB/s		0.00 GB/s	
Shared loads	0.00 GB/s		0.00 GB/s	
Shared stores	0.00 GB/s		0.00 GB/s	
Global loads	198.91 GB/s		195.41 GB/s	
Global stores	17.49 GB/s		16.77 GB/s	
Total	263.26 GB/s	3	212.18 GB/s	1
L2 Cache				
Reads	38.04 GB/s		195.48 GB/s	
Stores	30.51 GB/s		17.13 GB/s	
Total	68.55 GB/s	4	212.61 GB/s	5
Device Memory				
Reads	13.83 GB/s		9.46 GB/s	
Stores	24.44 GB/s		7.26 GB/s	
Total	38.27 GB/s	3	16.72 GB/s	1

5.2.2.3 Double versus Single Precision

In the testing process, it was found that double precision was needed, and double precision will also require more GPU cycles compared to single precision. In this section we will look into the impact of using double precision with a range of different GeForce and Tesla GPUs. Since GeForce GPUs have a smaller amount of double precision units than Tesla GPUs, we should expect that GeForce GPUs suffers more from using double precision than Tesla GPUs.

The tests results are shown in Figure 5.26 on page 91 for a variety of different Tesla and GeForce GPUs. 576 - 1 609 218 number of finite elements was used, and the simulation was running for 1000 frames. Detailed timing results are shown in Appendix E.3 on page 136.

From the results we can see that the ratio between single and double precision is always higher when using a Tesla GPU compared to a GeForce GPU with the same hardware architecture, which is expected. However the Tesla C2070 and the GeForce GTX-760 are more or less experiencing the same performance decrease due to double precision.

5.2.2.4 Frame Rate

The fracture simulation implemented in this project, requires a lot of calculations due to the amount of finite elements, and the real-time requirement of the snow simulator is no longer possible when using large mesh sizes.

In this section, we will first look at the achieved number of frames per second (FPS) for different mesh sizes. When running the tests, the simulation was running for 1000 frames. and rendering of the snow particles was turned off, and the terrain shader type was set to *Simple*, in order to increase the frame rate. In Table 5.9 on page 91 the results are shown when running the simulation on Workstation 1, and Table 5.10 on page 92 shows the frame rate for workstation 2, and due to the memory limit on the GTX-480 GPU, the largest mesh sizes could be simulated,

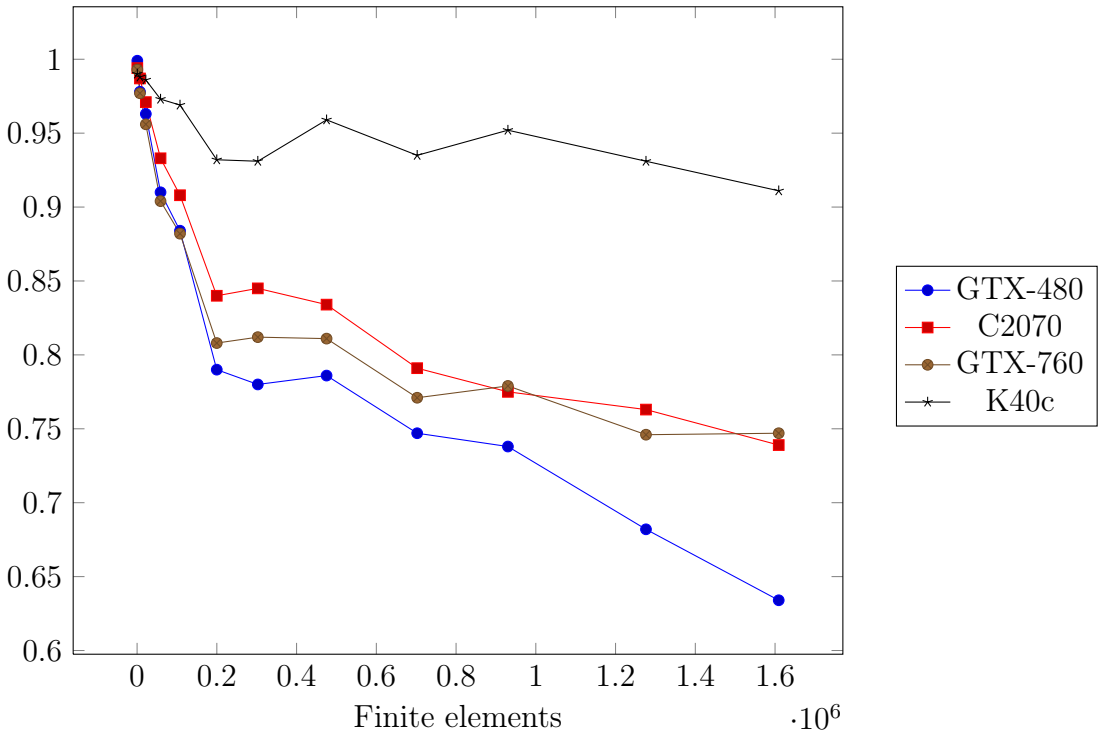


Figure 5.26: Performance penalty of using double precision, GeForce GTX-480, GeForce GTX-760, Tesla C2070 and Tesla K40c, separately

where workstation 1 simulates up to 13 275 637 finite elements, and workstation 2 simulates up to 45 259 256 finite elements.

The number listed in the parentheses in both tables, are simulations that was performed without optimal numbers of threads per block, namly the $\{4, 4, 4\}$ combination listed in Table 5.5 and 5.6 on page 85. And for small simulations, the total time difference results only in a couple of seconds, but when the number of finite elements are increased, this minor misconfiguration results in simulation times with a time difference about 8 and a half minute.

Table 5.9: Frames per second using workstation 1

Elements $\{x, y, z\}$	Total elements	Average FPS	Total time (seconds)
$\{99, 6, 99\}$	58 806	28.69 (26.7)	34.86 (37.4)
$\{199, 12, 199\}$	475 212	14.00 (11.7)	71.41 (85.4)
$\{299, 18, 299\}$	1 609 218	6.10 (4.6)	163.88 (215.2)
$\{399, 25, 399\}$	3 980 025	2.80 (2.1)	357.14 (485.8)
$\{499, 31, 499\}$	7 719 031	1.63 (1.1)	613.94 (884.4)
$\{599, 37, 599\}$	13 275 637	0.94 (0.7)	1068.9 (1358.8)

Table 5.10: Frames per second using workstation 2

Elements $\{x, y, z\}$	Total elements	Average FPS	Total time (seconds)
$\{99, 6, 99\}$	58 806	39.58 (37.49)	25.27 (26.68)
$\{199, 12, 199\}$	475 212	16.49 (15.41)	60.63 (64.903)
$\{299, 18, 299\}$	1 609 218	6.25 (5.81)	160.09 (171.98)
$\{399, 25, 399\}$	3 980 025	2.80 (2.56)	356.94 (390.95)
$\{499, 31, 499\}$	7 719 031	1.69 (1.40)	591.89 (716.77)
$\{599, 37, 599\}$	13 275 637	0.97 (0.80)	1020.88 (1244.35)
$\{699, 43, 699\}$	21 009 843	0.60 (0.52)	1653.43 (1910.15)
$\{799, 50, 799\}$	31 920 050	0.39 (0.34)	2554.73 (2978.36)
$\{899, 56, 899\}$	45 259 256	0.28 (0.24)	3558.41 (4074.65)

5.2.2.5 Max Register per Thread

When running the simulation with the Nvidia visual profiler to examine the performance results, it was discovered that the achieved GPU occupancy[1] was very low, around 30%. The reason why the achieved occupancy was that low is due to the number of registers that each thread uses.

The GPU has a finite number of registers available for each block to use, and if each thread within the block uses a lot of registers, the GPU is limited to execute fewer blocks simultaneously per SM. e.g. a GPU can either execute one block which uses all registers available, or the GPU can execute 5 blocks simultaneously where each block uses 20% of the available registers.

In my case, on the Tesla C2070, the maximum number of registers available for a block is 32758, and the *solve_global_displacement* kernels uses 46 registers per thread, and within each block there are 512 threads, and the profiler states that each block uses 24576 registers[2]. This will then limit the GPU to maximum execute 1 block simultaneously per SM. But the maximum numbers of registers available for each thread can be limited on compile time. And in Table 5.11 on page 92 the results are displayed, and we can see that when limiting the number of registers available for each thread, we can execute more blocks simultaneously and therefore achieves a better occupancy. However, we can also see that a higher occupancy does not results in better execution time for both kernels, and in our case, it is more or less best to not limit the number of registers per thread.

It should also be noted that these numbers are only for the *step1* kernel for both *'solve global displacement'* and *'propagation fractures'*, since the Nvidia visual profiler limits the debugging to one kernel concurrently. And the simulation was only running for 20 time steps, since the debugging adds a lot of overhead, and makes debugging very time consuming. However, in my opinion, I think that the general picture is well displayed in the results.

[1]Occupancy: Defined as the number of active warps divided by the maximum numbers of active warps

[2]With 46 registers per thread and 512 thread per block, the total number of registers per block should be $46 \times 512 = 23552$. To obtain a total of 24576 each thread must use 48 registers, but each thread uses probably 2 extra registers for thread identification

Table 5.11: Limiting the numbers of registers

Tesla C2070					
		Displacement Kernel		Fracture Kernel	
Max Registers	Threads/Block	Occupancy	Time[3]	Occupancy	Time
16	512	88.6%	694.73us	93.6%	3.4513ms
16	1024	61.0%	628.30us	64.0%	3.4513ms
32	512	61.4%	196.85us	62.7%	1.7029ms
32	1024	62.9%	226.59us	63.9%	1.7386ms
unlimited	512	31.7%	**181.24us**	32.3%	**1.0292ms**
Tesla K40c					
16	1024	89.1%	520.47us	95.3%	2.7993ms
32	1024	87.4%	279.17us	92.1%	1.6623ms
64	1024	47.4%	265.93us	47.2%	956.91us
128	1024/512[4]	47.4%	266.02us	24.1%	**955.98us**
unlimited	1024/512	47.2%	**265.42us**	24.0%	970.80us

5.2.2.6 CPU Version

A CPU version was also implemented in this project, to compare the GPU and the CPU, but the CPU version was only implemented to do the same calculations as the CUDA kernels, and no visualization was performed. The CPU version was also parallelized with OpenMP to fully utilize the power of the CPU. The GPU calculation times was obtained by using the Nvidia *nvprof* tool for linux, where you can find the average execution time for all kernels. The total execution time was then found by summing the average execution time for all 16 kernels, and multiplying by the number of time steps.

The results are shown in Table 5.12 and Figure 5.27 on page 94 for running the simulation for 100 time steps. And again, when testing the Tesla C2070, the rendering GPU in workstation 1 limited the number of finite elements that we could simulate, however the performance of the C2070 and the K40c is almost identical in this scale. And the reason why the K40c is not exceeding the C2070 in performance is due to that the L1 cache is not being used, and significantly speedup should be expected if the fast on-chip shared memory is used.

Also a comparison with the CPU version running sequential was performed, and the results are displayed in Figure 5.28 on page 94, where you have the sequential version running on the i7-4771, and the best execution times obtained on the GPU and parallel CPU is also added in the plot. And we can clearly see that parallelization gives huge performance increase in both cases.

[3] Average execution time for a single time step

[4] 1024 threads was used for the displacement kernel, and 512 threads was used for the fracture kernel

Table 5.12: Execution time for 100 iterations for different CPUs and GPUs

Finite elements	i7-4771	i7-3770	i7-870	Tesla C2070	Tesla K40c
58 806	1.86 s	2.24 s	3.26 s	0.56 s	0.49 s
475 212	14.72 s	17.22 s	23.60 s	4.19 s	4.01 s
1 609 218	58.48 s	65.67 s	73.12 s	13.62 s	14.06 s
3 980 025	158.63 s	168.72 s	179.70 s	33.38 s	33.85 s
7 719 031	307.69 s	328.02 s	404.52 s	59.87 s	57.39 s
13 275 637	517.02 s	583.34 s	625.84 s	103.99 s	100.62 s
21 009 843	828.85 s	922.16 s	926.36 s	——	164.24 s
31 920 050	1278.03 s	1433.84 s	1377.32	——	255.08 s

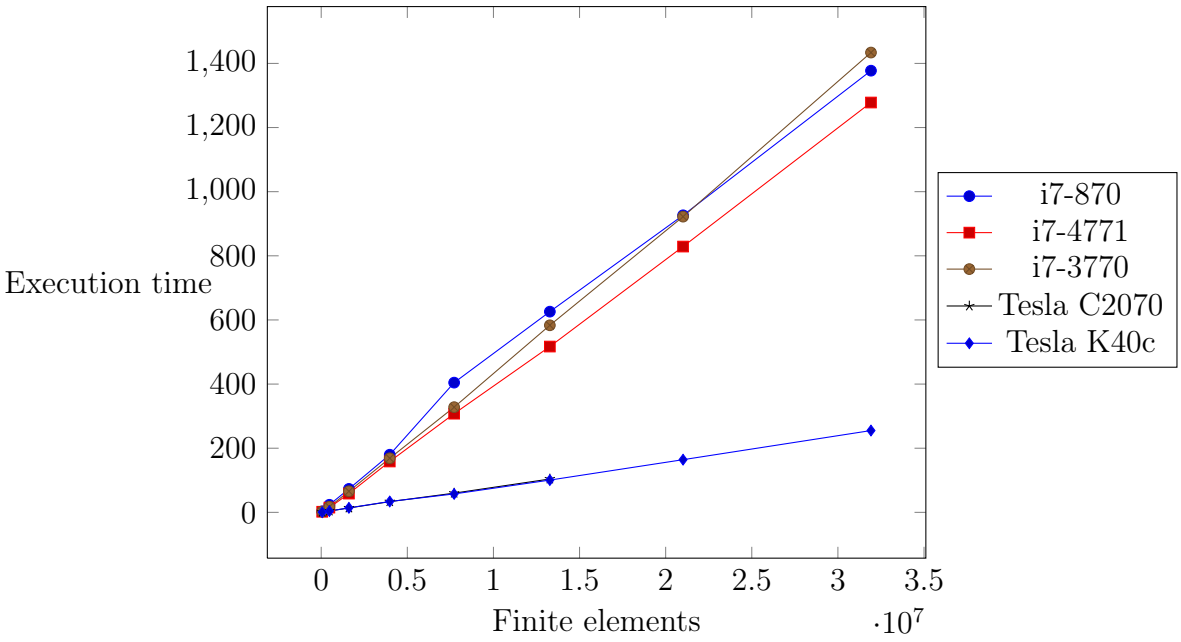


Figure 5.27: Execution time for different CPUs and GPUs

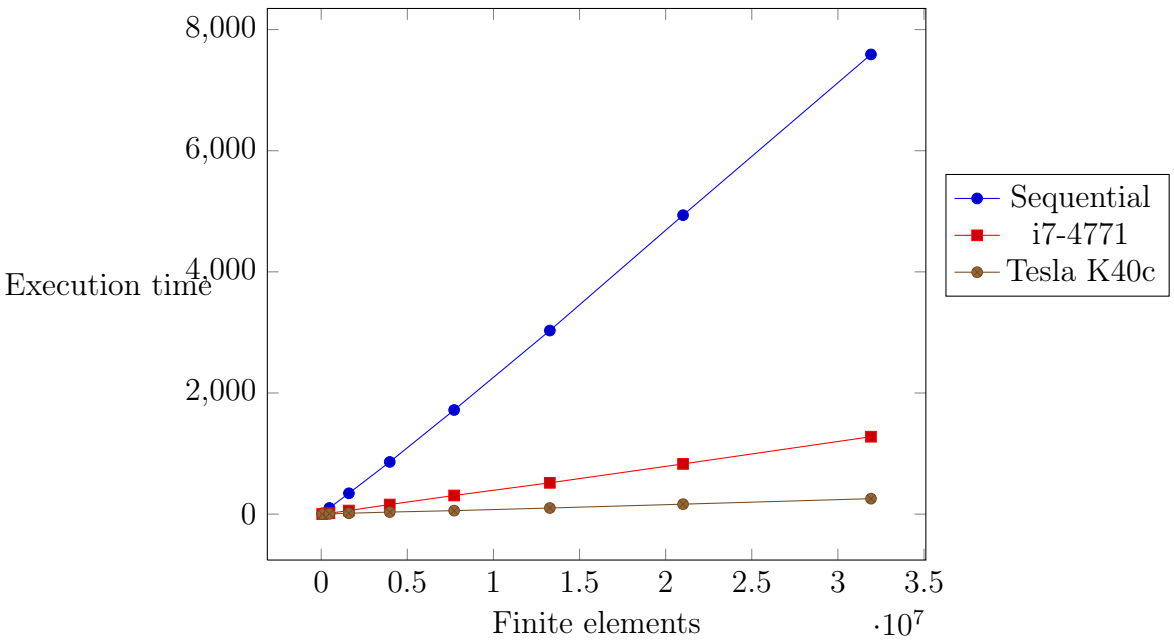


Figure 5.28: Sequential versus parallel CPU and GPU

Table 5.13: Kernel average execution time

	Kepler		**Fermi**	
Num elements	Displacement	Fractures	Displacement	Fractures
58806	0.99 ms	3.83 ms	0.82170 ms	4.75442 ms
475 212	8.25 ms	31.62 ms	6.29974 ms	35.4745 ms
1609218	29.36 ms	110.04 ms	20.9916 ms	114.933 ms
3 980 025	70.01 ms	276.68 ms	51.0379 ms	282.605 ms
7 719 031	123.95 ms	449.96 ms	91.67900 ms	508.329 ms

5.2.2.7 Fermi Vs Kepler

In the previous Section 5.2.2.6, we compared the total execution time for different CPUs and GPUs. And from the results in Table 5.12 on page 94 we can see that the Tesla K40c is not fastest in all cases.

The test results displayed in Table 5.12 are the results after running the simulator for a 100 time steps. The reason why a 100 time steps was selected was caused by the slow execution time on the CPU, and running 1000 time steps would have taken several hours. But since the GPU results are somewhat inconsistent, where the Fermi architecture has best execution time for some problem sizes, the test was performed again with the GPUs only, but running for 1000 time steps. The results are shown in Table 5.13, and the results is now more detailed than in the previous section. In the previous section the results was obtained by summing the average execution time for all kernels, then multiplying by the number of time steps. But since the two majority of kernels are very different, we have separated the kernels in these results, and summed the average execution time for the *displacement* kernels and *fracture* kernels respectively.

From the results in Table 5.13 we can now see a more consistent result, where the Fermi GPU (C2070) have a better execution time than the Kepler GPU (K40c) when running the *displacement* kernels, and the Kepler GPU is always faster than the Fermi GPU when running the *fracture* kernels.

Chapter 6

Discussion

In this chapter, we will discuss various aspects of this project, which includes; some issues with the mesh generation process, and the calculation of the global displacement, and the fact that our spring constant is not known. We will also look into some floating point accuracy issues that was discovered in the testing process, which was shown to be an issue when calculating the shear stress in the snow structure.

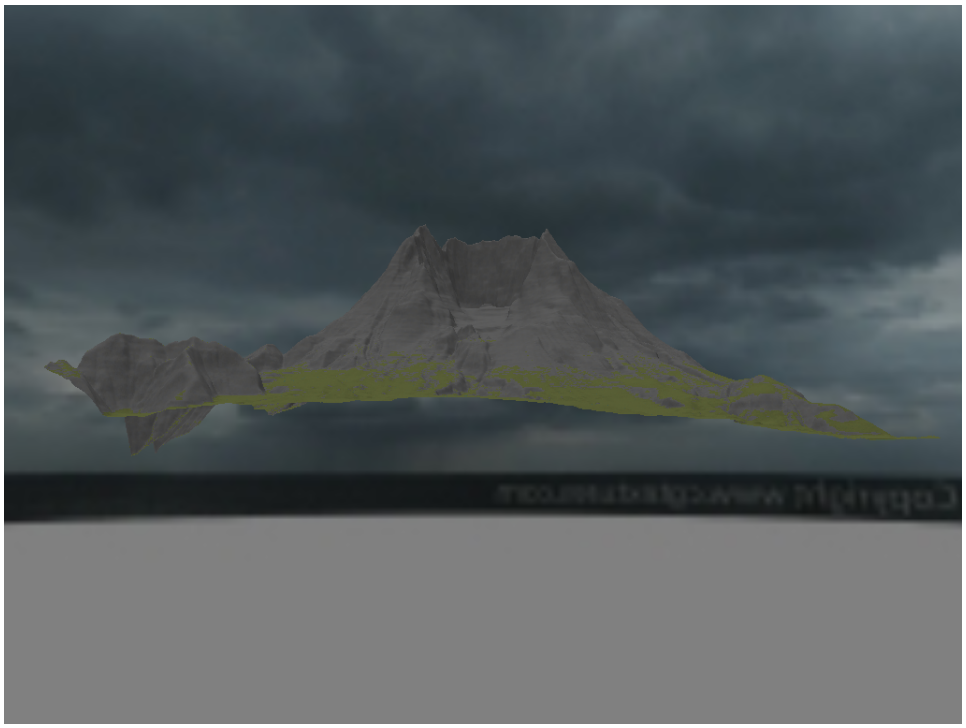
We will also discuss some performance aspects of the simulator, where we look into the achieved GPU occupancy, and the impact of ECC memory, and lastly look into reasons why some of the kernels are performing better on the Fermi architecture compared to the Kepler architecture.

6.1 Mesh Generation

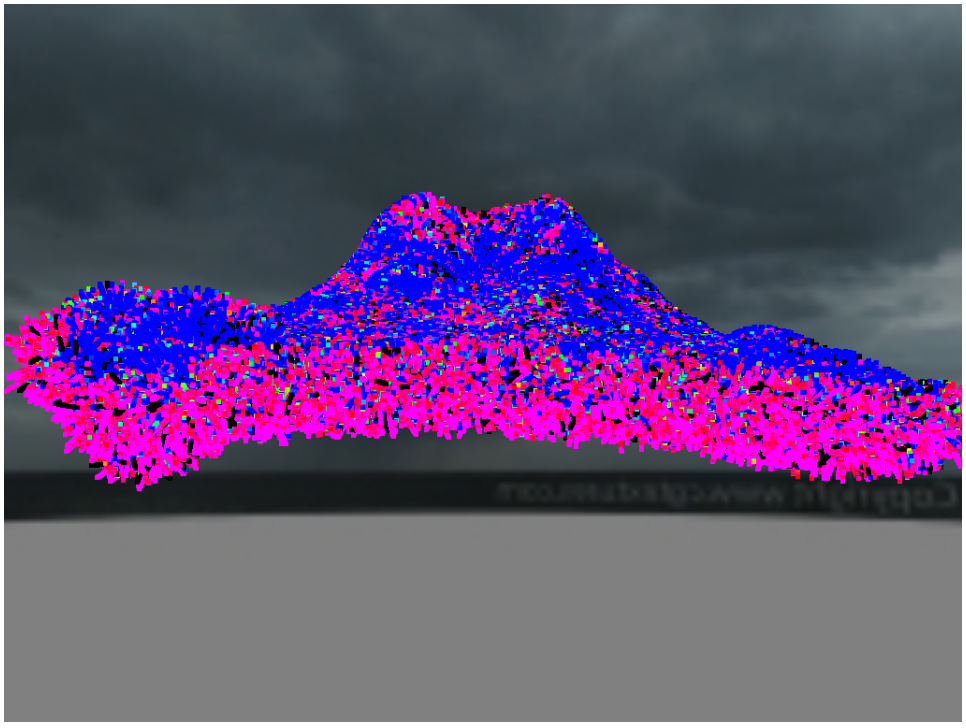
The mesh generation process is implemented such that the normal vector is calculated for each vertex in the terrain, resulting in a mesh with boxes lying parallel with the slope (Figure 4.4 on page 45). But after this approach was implemented, it was discovered that this method had a huge drawback. First of all, this method requires that the terrain heightmap is continuous, and therefore the usual terrains used in the snow simulator cannot be used. If the usual terrains are used, parts of the mesh will be located below the actual terrain, which is shown in Figure 6.1 on page 97, and in Figure 6.1b, the exterior nodes are visualized for all the finite elements, and approximately all the purple nodes are located below the actual terrain.

Another drawback with the implemented mesh generation is that narrow valleys cannot be generated (Figure 6.2 on page 98). In the figures, the visualization is set to visualize the density, where each finite element has equal mass. And first we can see in Figure 6.2a, the mesh is successfully generated. But in Figure 6.2b, the nodes of the finite elements are merging into each other, and in Figure 6.2c the valley is so narrow that the nodes are located below the terrain.

The reason why the mesh generation process fails when the terrain contain narrow valleys, is due to that the nodes located in above layers are created a distance dy in the normal vector direction. And this distance can cause normal vector to cross each other, and is displayed in Figure 6.3 on page 99, where the vertices in the above layer will be created with coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2).

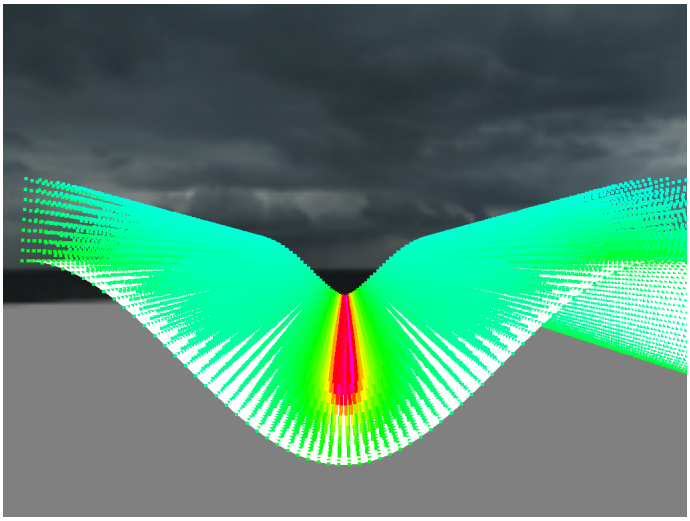


(a) Terrain without mesh

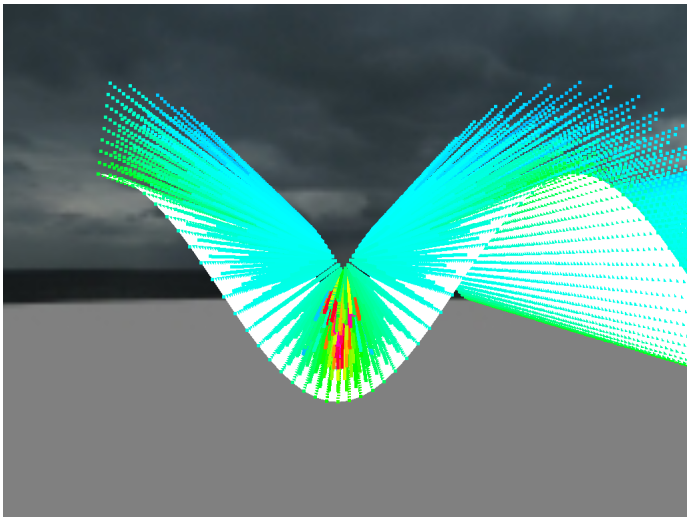


(b) Terrain with mesh located underneath the actual terrain

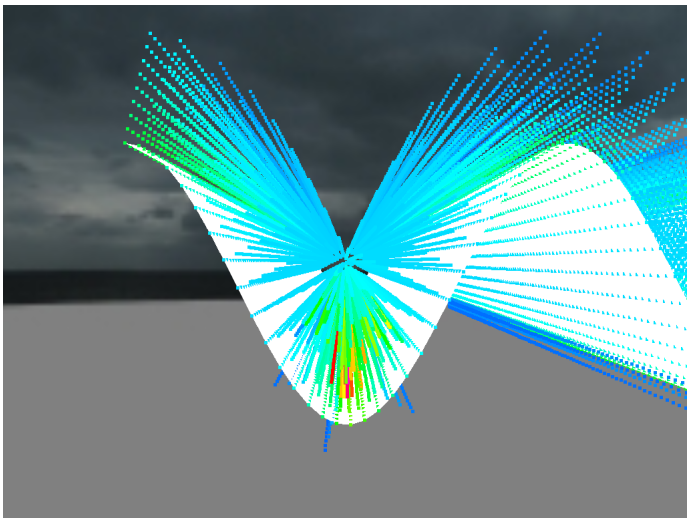
Figure 6.1: Automatic mesh generation creates nodes underneath the terrain



(a) terrain heightmap function = $5cos(0.1x)$



(b) terrain heightmap function = $5cos(0.15x)$



(c) terrain heightmap function = $5cos(0.2x)$

Figure 6.2: Narrow valley mesh generation

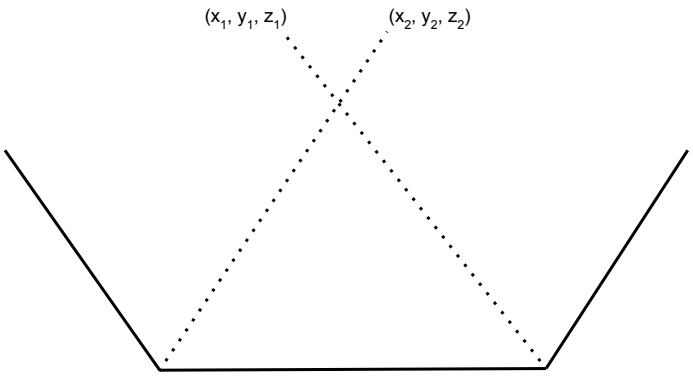


Figure 6.3: Mesh generation failure

6.2 Global Displacement Calculation

The calculation of the global displacements is based on the finite element method, which is used to assemble a system of equations, and is then solved by the iterative method called *'Successive over-relaxation'*. This method is used for solving system of equations in the form of:

$$Ax = b$$

Where A is a $n \times n$ matrix, and x and b are vectors of size n. And the local system of equations for a given finite element number i is shown in Equation 4.6 on page 46. And then further to solve the global system of equations, this local system will have to be assembled into a new matrix which represents the total system containing the equations for each finite element.

But the assemble of the global matrix was never explicitly performed, and in the implementation, one thread is spawned per finite element, and the matrix which is solved is the local stiffness matrix shown in Equation 4.6 on page 46. And this is most likely the reason to the issue present in the simulation when the global displacement.

The issue present when solving the global displacement, is that we cannot use a high relaxation factor with the *Successive over-relaxation* method. The relaxation factor is a parameter to the method that speeds up the convergence rate, and when we use a relaxation factor slightly above 1, the snow structure is literally torn apart (Figure 5.1c on page 66). And to obtain stable results we have to use a relaxation factor around 0.1, which increases the number of iteration needed to obtain the solution.

The Successive over-relaxation method is outlined in Listing 4.1 on page 46, and the relaxation factor ω is shown on line 14. The relaxation factor is used in this method to either focus the iteration in the direction of $(b_i - \sigma)$, which is the residual, or focus the iteration on ϕ_i which is the current guess. Therefore assuming that our iterative method is convergent, we should be able to use relaxation factors above 1.0 in order to speed up the convergence, but this is not the case in our simulation. In all tests performed in Chapter 5, we use a relaxation factor of 0.1, and higher relaxation factor causes instabilities. But it should be noted that these instabilities are only visible when calculating the stress in the snow. This is due to that the Young's modulus is tens of Mega pascal, and therefore when calculating the stress caused by a given strain, by the the formula:

$$\sigma = E\,\epsilon$$

The slightest instabilities in the displacement calculation (used to determine the normal and shear strain), will have a huge impact on the calculation of the stress. So when using a relaxation factor around 1.0 we can see that the stress level is highly unstable, but the vertices spatial location is more or less unchanged.

6.3 Spring Constant

The displacement calculation in the simulation, is based on Hooke's law, which is used for calculating the energy stored in an elastic spring, when it is either compressed or decompressed a given distance u. And the spring constant is essential to this law, but unfortunately this is not known, and therefore we cannot know if our results are anything near a realistic case.

But it may be that the spring constant used in the test are somewhat physical correct, due to the following reasons:

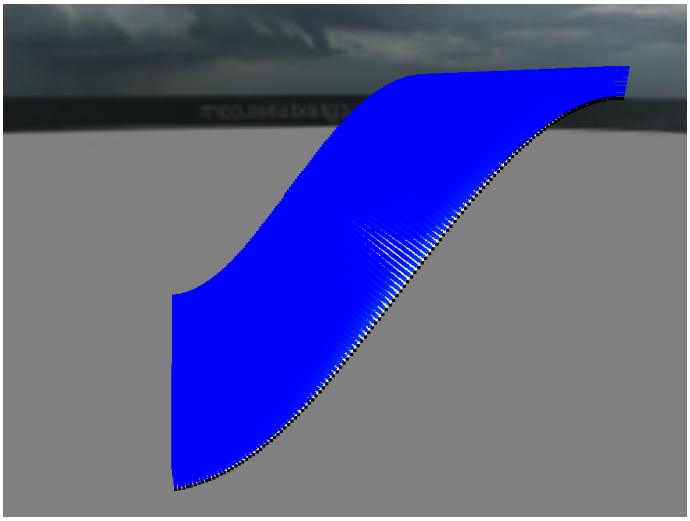
- When simulating homogeneous snow with a Young's modulus of 12 MPa, which is the around the lowest value of Young's modulus that was found by Sigrist[21], we obtain a solution where our calculated *Energy ratio* is approximately equal to zero across the entire domain (Figure 6.4a on page 101).
- And when simulating homogeneous snow with a Young's modulus around the highest possible value measured by Sigrist of 60 MPa, we obtain a relative high level of *Energy ratio* in the middle of the slope and we can see that fractures obtain enough energy to propagate a distance $da = 10^{-11}m$ once (Figure 6.4b on page 101).
- Lastly, when we simulate heterogeneous snow we can see that we have considerably higher *Energy ratio* across the entire slope, and we can also see that the fractures are propagating continuously (Figure 6.4c on page 101).

And all above cases are simulated with the same spring constant $k = 10^4\rho\ N/m$.

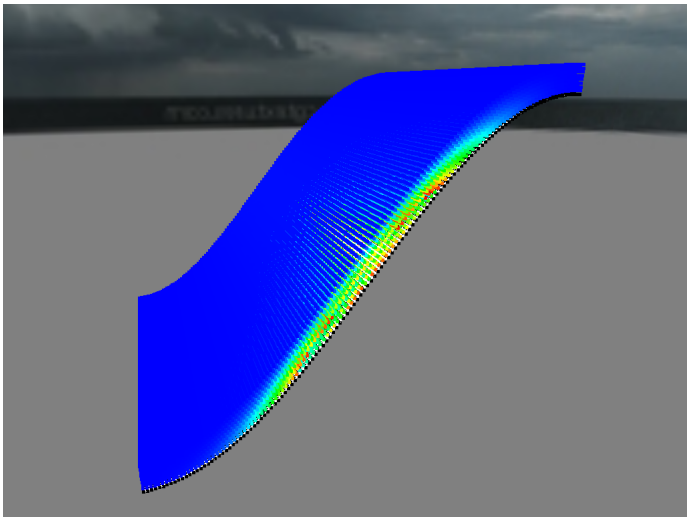
6.4 Shear Stress Accuracy

In the testing process, it was found that the shear stress calculations was highly unstable when using single precision. A small sample program was then implemented in order to test the normal and shear stress calculations between two vertices v_1 and v_2 with a displacement u_1 and u_2. The sample program calculations was tested with single and double precision, and it was found that the normal stress calculation did not experience any gaps in the calculations when small displacement was applied, and the difference between single and double precision is minor. However the shear stress calculation was experiencing huge gaps, and large difference between single and double precision. (test program listed in Appendix H.5).

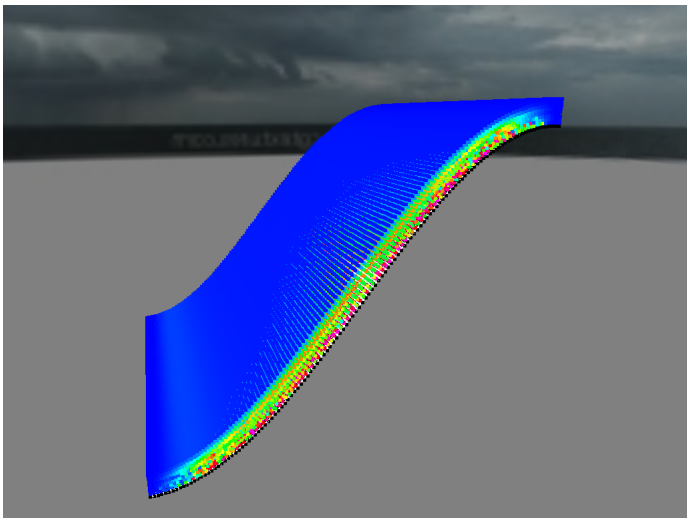
As we can see in Table 6.1 on page 102, the resulting stress does not vary that much between single and double precision, and the stress is also calculated accurately for very low displacements all the way down to 10^{-12}. However, as we can see in Table 6.2 on page 102, that the difference between single and double precision when calculating the shear stress are much greater, and we can also see that when using single precision, we do not obtain any shear stress for a displacement of $u_1 = \{0, 10^{-4}, 0\}^T$.



(a) Homogeneous snow, $E = 12MPa$



(b) Homogeneous snow, $E = 60MPa$



(c) Heterogeneous snow, $E = 1.89\rho^{2.94}Pa$

Figure 6.4: Energy ratio for different kind of snow using same spring constant

Table 6.1: Normal stress calculation

u_1	u_2	Normal stress (float)	Normal stress (double)
$\{0,0,0\}^T$	$\{0,0,0\}^T$	-nan	-nan
$\{10^{-1},0,0\}^T$	$\{0,0,0\}^T$	1200000.000000	1200000.017881
$\{10^{-2},0,0\}^T$	$\{0,0,0\}^T$	120000.000000	119999.997318
$\{10^{-3},0,0\}^T$	$\{0,0,0\}^T$	12000.000977	12000.000570
$\{10^{-4},0,0\}^T$	$\{0,0,0\}^T$	1200.000000	1199.999970
$\{10^{-5},0,0\}^T$	$\{0,0,0\}^T$	120.000000	119.999997
$\{10^{-10},0,0\}^T$	$\{0,0,0\}^T$	0.001200	0.001200
$\{10^{-12},0,0\}^T$	$\{0,0,0\}^T$	0.000012	0.000012

Table 6.2: Shear stress calculation

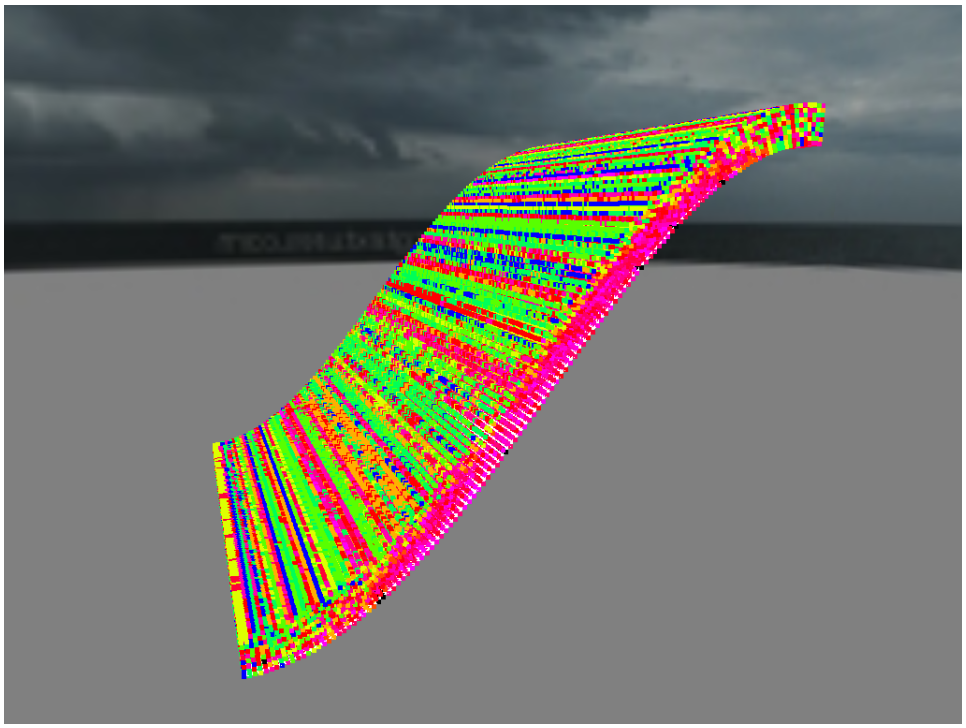
u_1	u_2	Shear stress (float)	Shear stress (double)
$\{0,0,0\}^T$	$\{0,0,0\}^T$	0.000000	0.000000
$\{0,10^{-1},0\}^T$	$\{0,0,0\}^T$	1196030.125000	1196023.847598
$\{0,10^{-2},0\}^T$	$\{0,0,0\}^T$	119938.906250	119995.997558
$\{0,10^{-3},0\}^T$	$\{0,0,0\}^T$	11718.750000	11999.996571
$\{0,10^{-4},0\}^T$	$\{0,0,0\}^T$	0.000000	1199.999970
$\{0,10^{-5},0\}^T$	$\{0,0,0\}^T$	0.000000	120.000005
$\{0,10^{-6},0\}^T$	$\{0,0,0\}^T$	0.000000	12.000533
$\{0,10^{-7},0\}^T$	$\{0,0,0\}^T$	0.000000	1.186117
$\{0,10^{-8},0\}^T$	$\{0,0,0\}^T$	0.000000	0.000000

Double precision was then applied to the simulation, and this gave good results for the shear stress calculations. In Figure 6.5 on page 103, the shear stress is visualized with single and double precision, and the simulation parameters are equal in both Figure 6.5a and 6.5b.

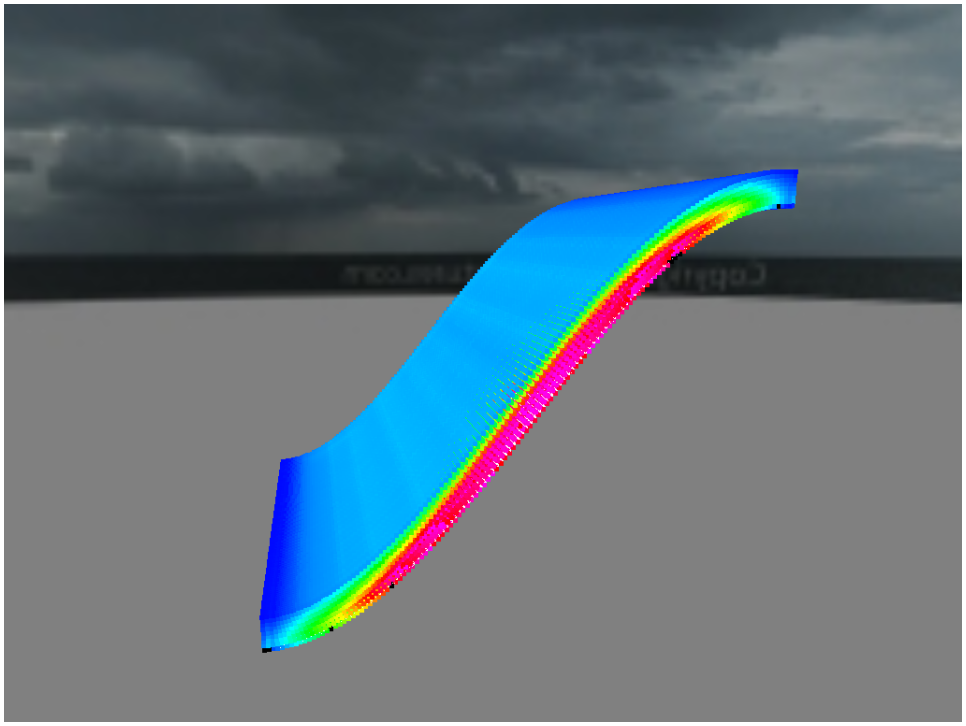
In Figure 6.5a, both number3 and number in Listing 6.1 are defined as float3 and float, respectively. And in Figure 6.5b they are defined as double3 and double, respectively.

Listing 6.1: Shear stress calculation

```
1  // Vars
2  float E;
3
4  // Global vertices
5  number3 v4, v8;
6
7  // Local displacement
8  number3 u4, u8;
9
10 // Vectors
11 number3 vec1 = v8-v4;
12 number3 vec2 = (v8+u8)-(v4+u4);
13
14 // Shear strain
15 number s;
16 s = acos(dot(vec1, vec2)/(length(vec1) * length(vec2)))*E;
```

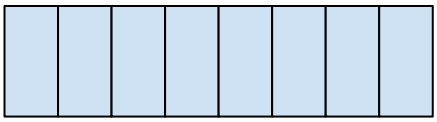


(a) Single precision

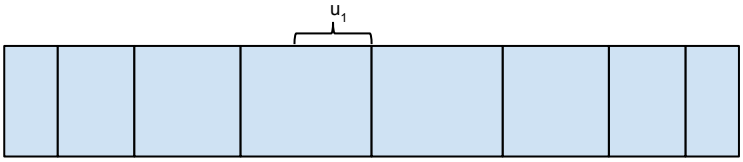


(b) Double precision

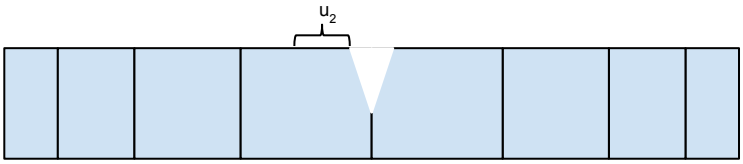
Figure 6.5: Shear stress precision



(a) Sequence of elements in original shape



(b) Elements being stretched, causing most displacement (u_1) and stress in the middle



(c) A fracture occurs which decreases the displacement $u_2 < u_1$, and therefore decreases the stress

Figure 6.6: Fracture displacement decrease

6.5 Calculation of Local Displacement

As discussed in Section 4.1.3, we have introduced an equation governing the local displacement for each element based on the length of the fracture. This idea came to mind due to that fractures act as a way to minimize the stress level in the material, and minimizing the stress is achieved by reducing the displacement.

The idea is shown in Figure 6.6, where we have a sequence of elements representing some material, and in Figure 6.6a the structure is shown in its original shape, then further in Figure 6.6b, the structure is being pulled outwards in both directions. This external pulling force will then cause most stress and displacement in the center of the structure (assuming a homogeneous material), and just before any fracture process occurs the structure has obtained a displacement of u_1 for the elements at the middle. Then suddenly a fracture process occurs shown in Figure 6.6c, which acts as the process which minimizes the stress present in the structure, by reducing the displacement around the fracture zone.

But in the implementation, a more simpler approach has been implemented which mimics this behaviour, which is shown in Equation 4.7 and 4.8. Where we simply have defined a relationship between the global displacement calculated by the SOR-method, and the so called local displacement which is used when calculating the stress present in the structure. And this equation reduces the local displacement with an increment in the fracture length.

The implemented solution may not be that physical correct, and a more complex approach could be implemented to achieve a more physical correct simulation. But in this project, a the simpler approach was selected due to time limitations, and also this serves as a good enough method to connect both the fracture simulation and the displacement simulation in this very first implementation.

6.6 GPU Occupancy

In Section 5.2.2.5, we looked into the achieved occupancy by varying the maximum numbers of registers available for each thread. By decreasing the number of registers that each thread can use, we allow more blocks to simultaneously be executed per SM. This is due to that each SM has a finite number of registers available, and these resources can either be used to execute 1 block, where each thread within the block can use tens or hundred of registers, or we can limit the number of registers such that the GPU can execute more blocks simultaneously, because it has available registers.

When programming on the GPU, high occupancy is often desirable, because having a high number of threads running simultaneously is known as the method to hide the memory latency on the GPU, and therefore obtain better execution time. But in our case, it was discovered that this was not the case.

This is probably due to that since we do not explicitly use shared memory, the fast on-chip memory will only be used as a L1 cache for a thread's local memory on the Kepler architecture, but Fermi will use this L1 cache for all memory operations. However the memory access pattern is not optimal. e.g when calculating the displacement for node 1 (see Figure 4.1 on page 43 for node numbering) of the finite elements, a memory request to i, $i+1$, $i+N_x$, and $i+N_x*N_z$ is performed. And since the total memory requirement for the displacement calculation does not fit within the shared memory on the GPU, we are probably experiencing a lot of cache misses when the on-chip memory is used as an L1 cache.

In the newest Kepler architecture, the on-chip fast memory can be configured to use a maximum of 48 KiB as a L1 cache, and when the numbers of SMs that a Nvidia GPUs has, is 15 for the most powerful GPUs today, this only sums up to $15*48 = 720KiB$, which is enough to store $(720*1024)/(3*4) = 61440$ float3 values, which is barely enough to store the displacement vectors for the smallest mesh sizes that we use, and in addition, when calculating the global displacement we need the global force vector as well, which has the same size as the displacement vector, and lastly, since the spring constant is depending on the density of the finite element, we also need to read the vertices of the 8 nodes for each finite element. Which triples the memory requirement. (propagate fractures kernels require even more).

So the GPUs do not have enough on-chip memory such that we can fit all our data within the L1 cache, and since we have a bad memory access pattern, we are probably experiencing a lot of cache misses. And this is probably why we achieve better execution time when we do not limit the number of registers available for each thread to use. Because in this case, the threads do not have to rely on the data being present in the L1 cache, because they will always have their data in the registers.

Although this limits the GPU to only execute one block per SM, which means that the first block will first read its required data, then perform its calculations, and finally write back the results, and then the second block can perform exactly the same operations. It shows that this is infact faster than allowing each SM to execute multiple blocks simultaneously, where each thread is depending on their data being present in the L1 cache.

Table 6.3: ECC impact on the Tesla C2070

Number of Elements	ECC on	ECC off
58 806	42.286482 s	34.617693 s
475 212	79.730415 s	71.097985 s
1 609 218	173.405227 s	163.464612 s
3 980 025	368.934885 s	357.009578 s
7 719 031	634.421283 s	613.505643 s
13 275 637	1068.561796 s	1046.984142 s

6.7 Error Checking & Correction Memory

After all the tests for Chapter 5 had been performed, it was notices that Error Checking & Correction Memory (ECC Memory) was turned off at workstation 1. And this was also informed by Eirik Myklebost[15], that it could have an impact on applications that are memory bound.

ECC memory is a type of memory that can detect and correct single-bit errors in the memory. So the word that is read from memory will always be the same word that was written to that location, even if one or more bits has been flipped to the wrong state. This type of memory is supported at systems where data corruption cannot be tolerated, like scientific computing. Hence you can change these settings at a Tesla GPU which is mostly used for scientific computing, but a GeForce GPU which are mostly used for gaming do not have this type of memory.

After it was detected that ECC was turned of at workstation 1, it was necessary to find out how big impact this has on the results, because the tests could not be executed once more due to time limitations. And as stated, ECC was turned off at workstation 1, but at workstation 2 it was turned on, and therefore the achieved speedup on the Tesla K40c should be somewhat higher. The total execution time results are shown in detail in Table 6.3 and Figure 6.7a on page 107, and in Figure 6.7b, we can see the ratio between the execution times by having ECC turned on and off, is becoming smaller and smaller with increasing problem size.

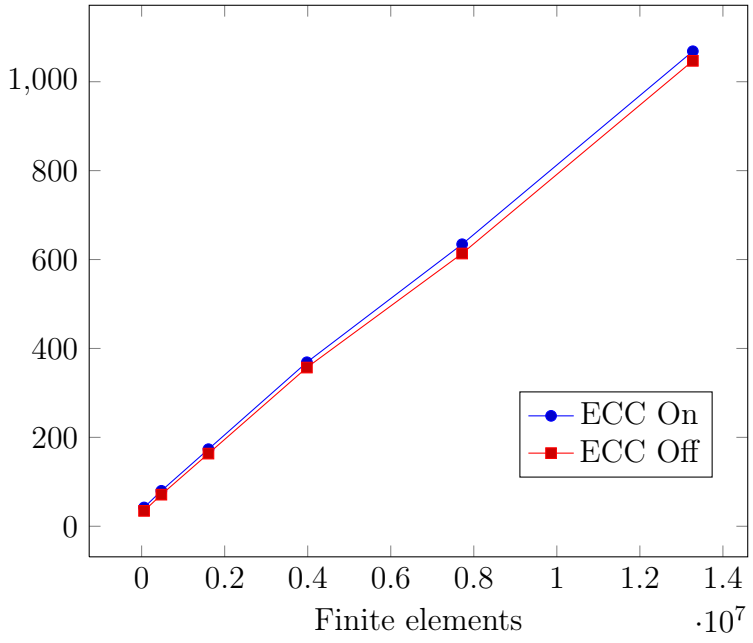
However, since this is a scientific simulation we should always have ECC enabled to obtain accurate simulation results.

6.8 Fermi Versus Kepler

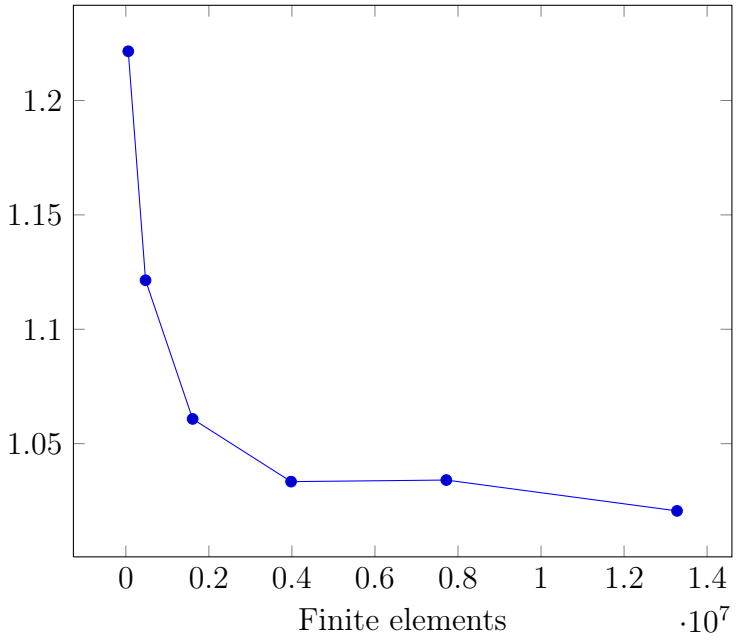
In section 5.2.2.7, we look at detailed performance results on the Fermi and Kepler GPU, and found out that the powerful Tesla K40c did not perform best in all kernels implemented in this project. But this is just shows a good example that you cannot just use better hardware, and assume that the speed will increase when you have a unoptimized code.

In Table 5.13 on page 95, we could see that the kernels calculating the displacement was performing better on the Fermi architecture, and the kernels calculating the fracture propagation was performing better on the Kepler architecture. And if we look more detailed on the hardware specification on the two GPU used (Table 6.4 on page 108), this comes as a big surprise.

However, as state earlier in Section 5.2.2.2, the usage of the fast on-chip memory has changed from the Fermi architecture to Kepler. The on-chip memory is located on each SM with a size of 64 KiB, and is used as the shared memory available on CUDA GPUs and as an L1 cache, and the programmer can decide on how these 64



(a) Total execution time with and without ECC by varying problem size



(b) Ratio between total execution times in Figure 6.7a

Figure 6.7: ECC impact on performance

KiB should be divided between shared memory and L1 cache, with the following options:

- 16 KiB shared, 48 KiB L1 cache
- 48 KiB shared, 16 KiB L1 cache
- 32 KiB shared, 32 KiB L1 cache (Kepler only)

But in this project, there have not been enough time to optimize the code, and therefore shared memory usage is not implemented, and the fast on-chip memory is only used as a L1 cache. And since Nvidia has changed how this cache is being used, where in the Fermi architecture, everything that is read from memory is being cached in this fast L1 cache. But in the Kepler architecture, only a thread's local memory is cached, and the local memory is only used iff a thread uses more registers than available. And as we looked at in Section 5.2.2.5, when using the Tesla K40c, the number of registers used for the *displacement* and *fracture* kernels are around 50 and 100, respectively. And the Kepler architecture supports up to 255 registers per thread, so the threads are not using any local memory, i.e. that the L1 cache is idle.

But the results shows that it is only the displacement kernels which performs better on the Fermi architecture. And this is most likely due to that the displacement kernels are only memory intensive, and the arithmetic is not so intensive, compared to the kernels calculating the fracture propagation, which has a lot of arithmetic and uses a lot of special arithmetic functions like *sqrt* and *acos*. And the number of special function units have been increased a lot on the Kepler architecture versus the Fermi architecture, 32 vs 4 per SM.

The CUDA compiler also has the option to disable any L1 cache usage (by compiling with *'-Xptxas -dlcm=cg'*), and by doing this we should see that the Kepler is performing better than Fermi for all kernels. The results without L1 usage is shown in Table 6.5 on page 109, and if we compare this with Table 5.13 on page 95 where the average kernel execution time is listed with L1 cache, we can infact see that the execution time is equal for the Kepler tests, because Kepler does not use the L1 cache. But the kernel execution times for the Fermi GPU is twice as large for the displacement kernel, and 32% - 39% larger for the fracture kernels when the L1 cache is disabled.

Table 6.4: Hardware specification on Tesla K40c and C2070

	Tesla K40c	**Tesla C2070**
Architecture	GK110B	GF100
CUDA cores	2880	448
Peak double precision floating point performance	1.43 Tflops	515 Gflops
Peak single precision floating point performance	4.29 Tflops	1.03 Tflops
Memory bandwidth (ECC off)	288 GB/sec	144 GB/sec
Memory size (GDDR5)	12 GiB	6 GiB
GPU Clock	875 MHz	1.15 GHz

Table 6.5: Kernel average execution time, L1 cache disabled

	Kepler		**Fermi**	
Num elements	Displacement	Fractures	Displacement	Fractures
58806	0.99 ms	3.83 ms	1.72 ms	6.37 ms
475 212	8.25 ms	31.62 ms	13.15 ms	46.90 ms
1609218	29.36 ms	110.44 ms	45.37 ms	157.95 ms
3 980 025	70.12 ms	267.73 ms	112.30 ms	395.12 ms
7 719 031	123.94 ms	449.92 ms	208.85 ms	702.28 ms

6.9 Finite Element Vibration

The simulation implemented in this project, is the result after several iterations with a trial and error approach. The very first implementation was quite different, and is shown more in detail in Appendix G. However, the initial FEM model was also quite different from the implemented model, where the first model was consisting of a set of tetrahedrons. And while this model was implemented, the method of solving the global displacement by using the SOR-method was highly unstable, and we could see the vertices bouncing in the snow layers.

While this issue was present in the simulation, various solutions was tried in order to remove the instabilities. And one of the solutions was to divide the global displacement calculation into several CUDA kernels, which was at the time a single kernel that calculated the displacement for all node of each finite element. And by dividing the code into several kernels that was executed sequentially on the GPU, a lot of race conditions was removed, and most of the instabilities was also removed.

But additional solutions was also tried in order to reduce the instabilities, and the other method tried was to instead of using the tetrahedron as the finite element, a set of cubes was also tried. And by changing the type of finite element, the system of equations that was required to solve was changed from the system shown in Equation 6.1 to the system shown in Equation 6.2. And if this gave any additional reduction in the instabilities is not known. But at the time, after visually inspecting the vibration present in the simulation, the vibration seemed less. However, any metric for calculating the amount of instability was never implemented, and if the vibration actually was reduced by changing the finite element from the tetrahedron to a cube is not known, and after the testing of the simulation was performed in Chapter 5, it was found that the relaxation factor used by the SOR-method has a huge impact on the instabilities. But at the time, it seemed that the cube was experiencing less vibration, and therefore this is the kind of finite element that is used in the implemented simulation.

$$k^{(i)} \begin{bmatrix} -3 & 1 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & 0 & 0 \\ 1 & -12 & \dots & 3 & 1 & \dots & 3 & 1 & \dots & 0 & 3 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 3 & \dots & -12 & 1 & \dots & 3 & 0 & \dots & 1 & 3 \\ 0 & 1 & \dots & 1 & -3 & \dots & 0 & 0 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 3 & \dots & 3 & 0 & \dots & -12 & 1 & \dots & 1 & 3 \\ 0 & 1 & \dots & 0 & 0 & \dots & 1 & -3 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & -3 & 1 \\ 0 & 3 & \dots & 3 & 1 & \dots & 3 & 1 & \dots & 1 & 12 \end{bmatrix} \begin{bmatrix} U_i \\ U_{i+1} \\ \vdots \\ U_{i+M_x} \\ U_{i+M_x+1} \\ \vdots \\ U_{i+M_x*M_z} \\ U_{i+M_x*M_z+1} \\ \vdots \\ U_{i+M_x*M_z+M_x} \\ U_{i+M_x*M_z+M_x+1} \end{bmatrix} = \begin{bmatrix} F_i \\ 4F_{i+1} \\ \vdots \\ 4F_{i+M_x} \\ F_{i+M_x+1} \\ \vdots \\ 4F_{i+M_x*M_z} \\ F_{i+M_x*M_z+1} \\ \vdots \\ F_{i+M_x*M_z+M_x} \\ 4F_{i+M_x*M_z+M_x+1} \end{bmatrix} \tag{6.1}$$

$$k^{(i)} \begin{bmatrix} -3 & 1 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & 0 & 0 \\ 1 & -3 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \dots & -3 & 1 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 1 & \dots & 1 & -3 & \dots & 0 & 0 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \dots & 0 & 0 & \dots & -3 & 1 & \dots & 1 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 1 & -3 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & -3 & 1 \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 1 & -3 \end{bmatrix} \begin{bmatrix} U_i \\ U_{i+1} \\ \vdots \\ U_{i+M_x} \\ U_{i+M_x+1} \\ \vdots \\ U_{i+M_x*M_z} \\ U_{i+M_x*M_z+1} \\ \vdots \\ U_{i+M_x*M_z+M_x} \\ U_{i+M_x*M_z+M_x+1} \end{bmatrix} = \begin{bmatrix} F_i \\ F_{i+1} \\ \vdots \\ F_{i+M_x} \\ F_{i+M_x+1} \\ \vdots \\ F_{i+M_x*M_z} \\ F_{i+M_x*M_z+1} \\ \vdots \\ F_{i+M_x*M_z+M_x} \\ F_{i+M_x*M_z+M_x+1} \end{bmatrix} \tag{6.2}$$

Chapter 7

Conclusion

In this thesis, we have investigated in using fracture mechanics, in order to predict avalanches. And during the time period of this thesis, it has been discovered that avalanche prediction is still a relative new field of research and also a extremely difficult field, due to the complex behaviour of snow. And from my knowledge there are not a lot of papers published in this area, regarding either computer simulations or experimental data, but there exists a few as we have look at in Chapter 3.

In this thesis, a introduction to fracture mechanics and the finite element method is given, and we also look into snow as a material and the properties giving snow its complex behaviour, and we also look at different kind of avalanches and the reason behind the fatal slab avalanches[1]. A FEM model has also been developed for simulating fracture propagation in the snow layers, where I also have developed a lot of different visualization methods, giving valuable visual feedback of avalanche prediction.

Even though this work may not be 100% physical accurate due to the unknown spring constant used, and work remains in order to fully integrate the implemented simulations into the snow simulator, I think that this work can be used in order to generate higher interest in the field of avalanche prediction. Where the current state of avalanche prediction has a lot of potential for improvement.

7.1 Avalanche Prediction

The initial goal for this thesis, was to simulate fracture propagation in the snow layers, and based on this, obtain a volume of the snow layers that would be included in an avalanche. But during this thesis, it was shown that the initial goal was difficult to obtain. This was caused by several different factors:

- During this thesis, I had to obtain a lot of knowledge within fracture mechanics, where I had no prior knowledge. And this field was learnt in great depth in order to fully understand the field. Where I started to learn the basic of fracture mechanics on the atomic level, and how the equations in fracture mechanics are created. And without this knowledge, I would not been able to adapt the equation of the **Energy Release Rate** to better fit the model created in this thesis. And this was necessary, because the basic equations in the field of fracture mechanics are equations suited for 2D problems, and

[1]The theory of *weak layers* seems promising, however it is yet to be proven

since our model is in 3D, we had to adapt the equation of how the energy release rate was calculated (Section 4.1.5).

- Knowledge about the Finite Element Method was also necessary to obtain, before the simulation could be implemented. And this was also a field where I had no prior knowledge. Fortunately, a book about FEM was found[6], where they had 1D examples of how the method could be used to model deformation, and this was then extended to 3D.

- The SOR-method is also used in this project, but the method was slightly known before the project was started. But only in theory, and I had never had any hand-on experience with the method.

- It was also found out that there existed little work in the field of avalanche prediction, and therefore I had few examples to follow, and the simulation approach implemented has been created by myself, which was a time consuming process where various approaches was evaluated.

- Additionally similar work was found [22], where the behaviour of snow is very accurately modelled, and it was then found how complex it is to accurately simulate how snow behaves. And during this time, it was found that this behaviour had to be simplified in order to achieve any kind of simulation.

The above factors has contributed to a lot of challenges in this project, and the initial goal was too ambitious, and therefore, it was simply not enough time to create an algorithm for calculating the initial volume of avalanches.

But even though that the calculation of the avalanche volume is not implemented, I think that this work is very useful for predicting avalanches. Because when slab avalanches are released, they are always dangerous and could be fatal, and the question of how big they would be, may not be necessary to answer in order to save human lives. We simply have to answer the question of where and when they are triggered.

The above question of where and when avalanches are triggered, I think is possible to answer with my model iff physical validation is made, and in addition we need to acquire a lot of fracture data on different kind of snow types, e.g. *critical energy release rate*. However, I think that this work gives a valuable start for further work on this issue.

7.2 Performance

In this project, we have taken advantage of the powerful graphical processing unit (GPU), which has shown to be 5x faster than the fastest CPU I had available (Tesla K40c vs Intel i7-4771) for problem sizes above 4×10^6 number of finite elements, and for the smallest problem sizes of 5.8×10^4, the GPU was 3.8x times faster.

It should also be noted that this is not a comparison against a sequential CPU version. The CPU version was compiled with *gcc* with optimization level *-O3*, and in addition parallelized with *OpenMP*. Which all made a huge performance increase on the CPU version. However, the simulation should be expected to have a potential for performance increase, when running the simulation on a GPU with the Kepler architecture, when *shared memory* is used.

A short comparison with a sequential version was also made, where the GPU had a speedup ranging from 25-30x, which shows how important it is to run parallel code. It should also be noted that the CPU also got significant speedup when using multi-threading with OpenMP.

7.3 Future Work

Over the course of this project, it has been shown that to completely predict avalanches much work remain. And in the following sections we will discuss various possibilities of extensions and improvements.

7.3.1 Mesh Filling

In the initialization stage of the simulator, a set of vertices are generated, representing the snow layers. In addition, a *mesh_point* structure is initialized, which is equal to the number of finite elements. And the idea of this structure is to store data representing the properties of the snow present in each finite element, and then based on this data, obtain the Young's modulus and critical energy release rate.

This mesh filling algorithm was implemented in my specialization project[26], where temperature and humidity present in the air when each snow layer was filled was stored in this mesh_point structure. However, in this project, it was discovered that the mesh generation had to be reimplemented, and therefore the mesh filling algorithm was no longer valid. And this is therefore a part that needs to be implemented, where we need to fill the mesh_point structure with snow properties as the snow cover increases.

Currently, the mesh_point structure is being passed on as parameter to CUDA kernels responsible to find the Young's modulus and the critical energy release rate for each of the finite elements. But in this project, this mesh_point structure is ignored, and constant values are returned.

In my specialization project, the fill mesh algorithm was implemented such that the 3D mesh_point structure was compared with the 2D height map, containing the snow height across the terrain, in order to figure out which parts of the mesh that had been covered with snow. And since the snow layers vertices was simply a number of duplicates of the terrain vertices positioned above in Y-direction (Figure 3.4 on page 36), the comparison of the heightmap and the mesh_point structure was strictly forward. Each mesh point with coordinate $\{X, Y, Z\}$ had to perform a lookup in the snow heightmap with coordinates $\{X, Z\}$. But with the implemented mesh generation in this project, the comparison is no longer that easy.

As mentioned in Section 3.1.1.2, when snow particles collide with the terrain/snow the 2D heightmap containing snow height data across the terrain is incremented with a predefined *snow_growth_coefficient*. And it is this heightmap that are visualized when using the *simple, complex, and perlin* terrain shader types. So to fully integrate the 3D snow layer mesh with the snow buildup, I envision two different approaches, where both has different challenges.

Method 1: In this method, I envision a approach similar as in my specialization project, where we simply compare the heightmap and the finite elements. This will in my mind be the easiest approach, where we do not need to change any code for the visualization or the movement and collision of the snow particles. But the

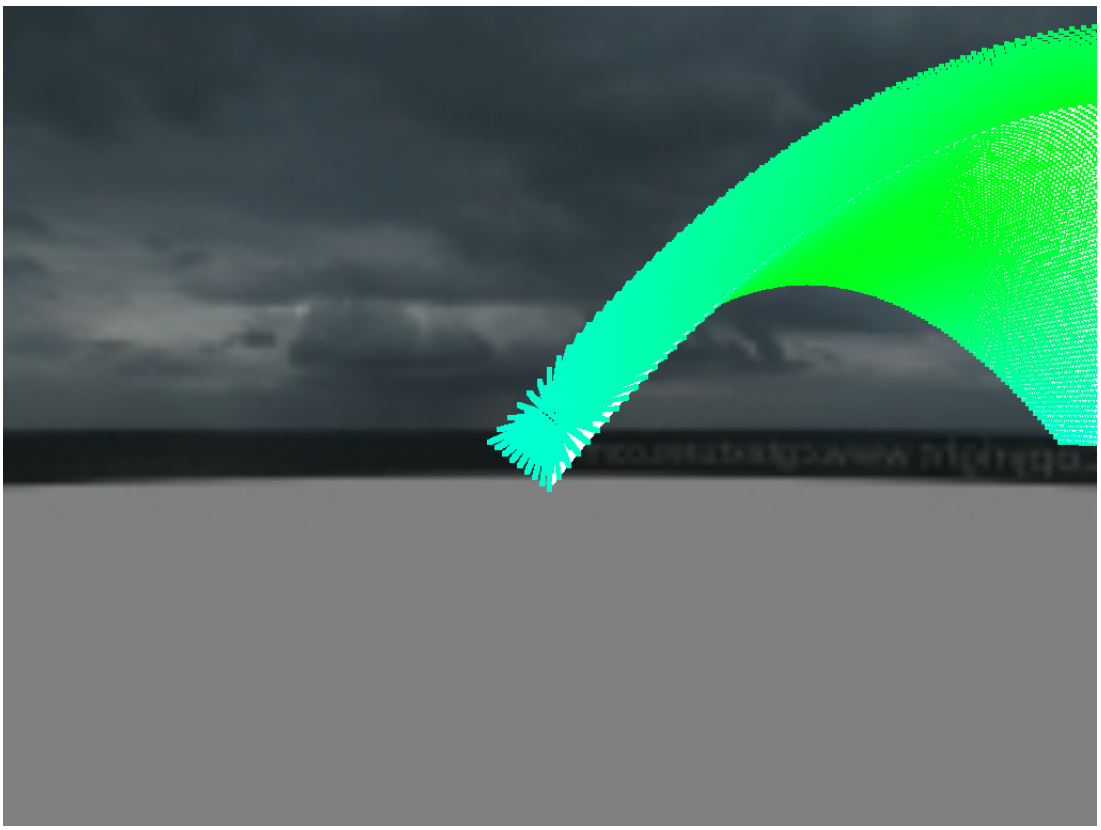


Figure 7.1: Issue with mesh filling method 01

downside of choosing this approach is that we limit the number of terrains that we can simulate, and the terrain displayed in Figure 7.1 would generate an issue.

This is because that the 2D heightmap can only grow in Y-direction, and therefore the finite elements located far to the left would never be filled.

Method 2: The other method that could be implemented in order to fill the finite elements, could be by directly bypassing the 2D heightmap, and reimplementing the collision detection of the snow particles. Each finite element could then have an indicator of the percentage fill, and when the snow particles collide with a finite element they would increase this fill percentage. However the negative effect of this, would be that many of the terrain shaders would have to be reimplemented as well, because they rely on the 2D heightmap representing the snow buildup. And in addition, it would be relatively difficult to render a smooth terrain. This is because the finite elements are relative large, and if we had only visualized each finite element, the terrain would be quite rough and we could have seen the cubes.

But in the long term, I personally think that this would be the best approach such that we could simulate all kinds of terrain, and in addition, we would no longer have the limitation that we cannot have any snow buildup which results in any snowdrifts as shown in Figure 2.15 on page 19.

7.3.2 Improve Mesh Generation

The mesh generation process has currently two weaknesses, one of them is discussed in Section 6.1, where we look at the reason why we cannot create meshes in narrow valleys, however I consider this only a minor issue which restrict us from simulating these kinds of terrains. Another issues which I would personally rank with higher

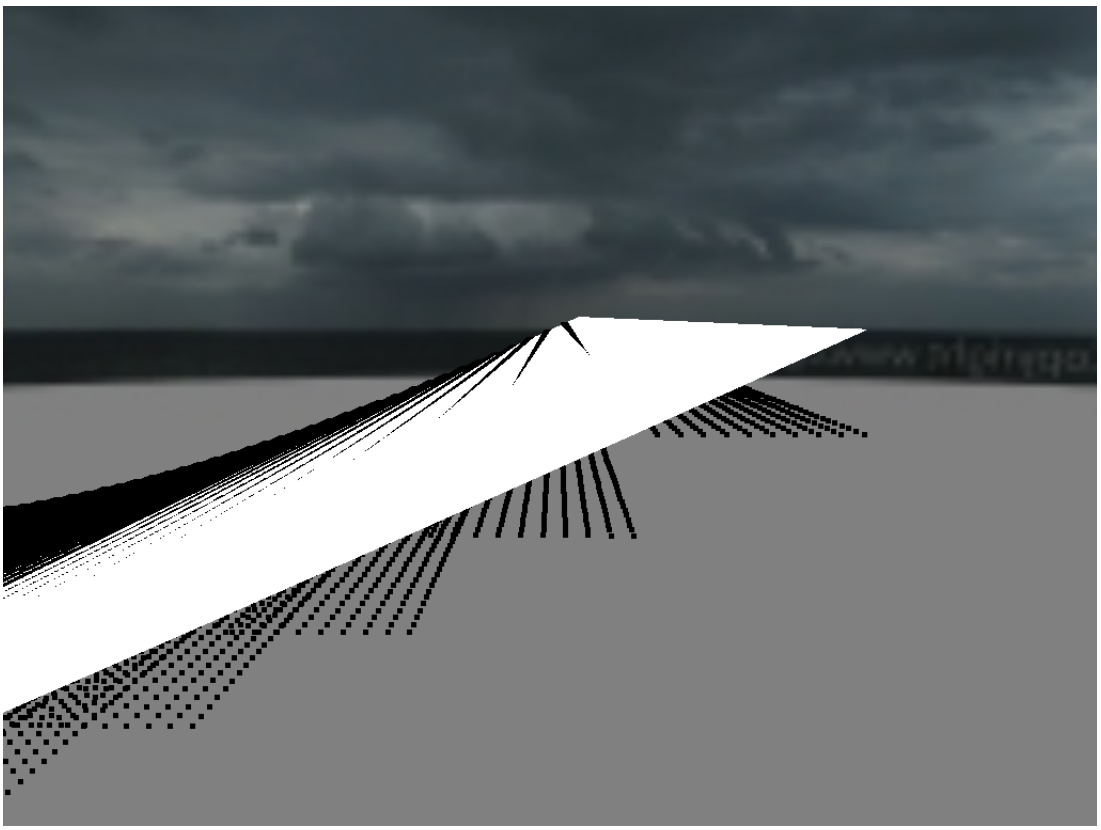


Figure 7.2: Mesh generation issue when using mesh deltas less than 1.0

priority is an issue when changing the accuracy of the mesh by reducing/increasing the *mesh_dx, mesh_dy, and mesh_dz.*

In my implementation, these variables is used such that if they are all equal to 1.0, the vertices for the bottom part of the finite elements, located in the lowest part of the snow layers, will correspond exactly with the terrain vertices. And this part works perfectly, however when using mesh deltas less than 1.0 there will be some vertices that will uses a incorrect terrain height values as displayed in Figure 7.2 on page 115.

The reason to this behaviour is due to, when the terrain heightmap is accessed in the mesh generation process, we have an X and Z floating number that are incremented by a value dx and dz, and right before accessing the heightmap array these are converted to integers. Resulting in that e.g. $\{1.0, 1.0\}$, $\{1.25, 1.0\}$, $\{1.50, 1.0\}$, $\{1.75, 1.0\}$ will access the terrain heightmap in the same location, resulting in the exact same height values.

This could then be fixed by using the plane formula which we could find by using a set of three vertices in the terrain, then calculate what the height should be for any point for that plane.

7.3.3 Avalanche Flow Simulation

Avalanche flow simulation has been implemented earlier by Øystein Eklund Krog[27], which simulated avalanche flow by using *Smoothed Particle Hydrodynamics* (SPH). Fortunately, his code was developed like a framework where he has separated the SPH simulations and the visualization, and therefore we should be able to use this framework in the snow simulator for avalanche flow simulation.

Avalanche flow simulations has the issue of the initial conditions, like where

in the terrain the avalanche is initiated and the volume of the avalanche. But if the avalanche flow simulation is integrated with this project, we would not longer have this issue. But it order fully integrate this project, with another project used for avalanche flow simulation, this project will have to be able to predict the total volume of an avalanche, then the particles used in SPH should be initialized within this volume.

Appendix A

Recreating Results

In this project, it has not been any focus point to create a fully polished simulator, where you have an user interface for changing which kind of terrain and snow type that are being simulated. So changing there parameters requires uncommenting code and recompiling the simulator. In the following sections we will look at which parts of the simulator code files that you need to change in order to perform the simulations yourself.

A.1 Setup

In Section 5.2.1 we listed different parameters that we used for all tests performed, which we will now look into how you can modify these yourself.

Applied Force: This is a vector of size equal to the total number of nodes in the simulation, and is initialized from line 124-132, in *TerrainSystem.cu*. Where the first line within the for-loop declares the x-component of the force, and next is the y-component, and lastly the z-component. But this is not changed in our simulations, and are therefore not necessary to change.

```
1 CUDA_SAFE_CALL(cudaMalloc((void**)&force, total_nodes*3*sizeof(float)));
2 float *global_force_data = (float*)malloc(total_nodes*3*sizeof(float));
3 for(int i = 0; i < total_nodes*3; ){
4     global_force_data[i++] = 0.f;
5     global_force_data[i++] = 200.f;
6     global_force_data[i++] = 0.f;
7 }
8 CUDA_SAFE_CALL(cudaMemcpy(force, global_force_data, (total_nodes*3*sizeof(float)),
      cudaMemcpyHostToDevice));
```

Spring constant: The spring constant is defined as a device CUDA kernel, and is found from lines 190-192 in *TerrainSystemKernels.cu*. However, the same constant is used for all simulation results, and are therefore not necessary to change.

```
1 /**
2 * Function for finding spring constant for a given finite element
3 **/
4 __device__ float spring_const(mesh_point *mesh_points, float3 *U, float3 *vertices
      , int id){
5     return 10000.f*find_density(mesh_points, U, vertices, id);
6 }
```

Young's modulus: Young's modulus is also defined as a device CUDA kernel and is found from line 173-176 in *TerrainSystemKernels.cu*.

```
1 /**
2 * Function for finding Young's modulus for a given mesh element
3 **/
4 __device__ float find_youngs_modulus(mesh_point *mesh_points, float3 *U, float3 *
      vertices, int id){
5         return 60.f * 1000000.f; // 60 MPa
6 }
```

Critical energy release rate: The Critical energy release rate is defined as a device CUDA kernel and is found from line 181-185 in *TerrainSystemKernels.cu*.

```
1 /**
2 * Function for finding the critical energy release rate for a given finite element
3 **/
4 __device__ float find_critical_energy_release_rate(mesh_point *mesh_points, float3
       *U, float3 *vertices, int id){
5     float k = 0.00042f * powf(find_density(mesh_points, U, vertices, id), 2.76f);
6     return (k*k)/find_youngs_modulus(mesh_points, U, vertices, id);
7 }
```

Stress release factor: The stress release factor is found as a constant float defined at line 30 in *TerrainSystemKernels.cu*. But this is not required to change because it is equal to 1000, as we have used in all results.

SOR factor: The relaxation factor is found in *Config.cpp* at line 97. But it is set equal to 0.1, for our simulations, so there should not be necessary to modify this either.

***'da'* factor:** The fracture propagation distance factor is found in *Config.cpp* at line 96. But it is set equal to 0.00000000001f, for our simulations, so there should not be necessary to modify this either.

Finite element mass: The mass of the finite elements are a local variable used when calculating the density for a finite element, and is found on line 164 in *TerrainSystemKernels.cu*, which is also constant for our simulations.

Changing terrain: To change the terrain used in the simulation is slightly more complicated. The terrain heightmap is initialized from line 735-789 in *Terrain.cpp*, and within these lines there are a lot of commented code, which should be uncommented in order to create the desired terrain.

In this project, we have used the following functions to describe the terrain:

- $y = 0$
- $y = ((x - 64)^2 + (z - 64)^2)/256$
- $y = -5cos(0.025x)$
- $y = -25cos(0.025x)$
- $y = cos(0.1x) + 0.01x * cos(0.1z) + 2$
- $y = 5(cos(0.09x + 1.5) + 0.045x)$

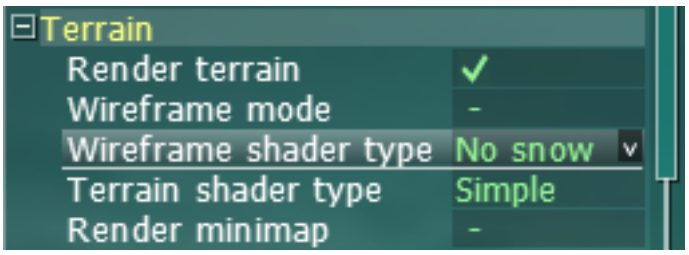


Figure A.1: Terrain section of simulation menu

It is also important that the terrain size is equal to 128, when using the above functions. This is because that the variables $X, Z \ \exists \ [0, terrain\_size]$. The terrain size is specified in the *data/config.txt* as the **map_size 128** property.

A.2 User Guide

In this section we will look briefly into the startup, the controls of the simulator, and how to change between the different visualization methods.

First the simulator is started by running the following command from the terminal, after you have changed into the directory containing the source files and make file:

./snow
or
./snow –default

When applying *–default*, the startup configuration menu step will be skipped, and the simulation starts with the default parameters.

Next is camera movement, where you use $\{W, S, A, D\}$ to move forward, backward, left, and right, respectively. And to look around, you need to hold the right mouse button and then move the mouse.

You can also use the space button to move directly upwards, and the LEFT-CTRL to move directly downwards. And in addition you can hold the LEFT-SHIFT to speed up any of the movement.

Lastly, as shown in Figure 4.11 on page 54, you have a screenshot of the simulation menu where you have the option to change between the different visualization methods, under the section *"Terrain"*, where the default startup visualization method is set to *"Simple"* (Figure A.1).

Appendix B

Finite Element Type

As stated in the start of Chapter 4, generating the mesh for a parallelepiped type of element would be easiest. But I think that this kind of element would create some issues with the implementation on shear stress. Where in Figure B.1 on page 121, the parallelepiped element is shown where the top vertices are being applied a force along the global Y-axis. And this would result in a displacement U_Y, and with my normal and shear stress calculations, the shear stress would be equal to zero, and only normal stress would be generated.

However, with the case of using the cube as the finite element as shown in Figure B.2 on page 121, our calculation would generate both a normal stress and shear stress.

And since shear stress/strain is defined as the following *'the change in angle between lines'*, we can infact see that the first case with the parallelepiped element should not generate any shear stress, and the latter case with the cube element should generate shear stress.

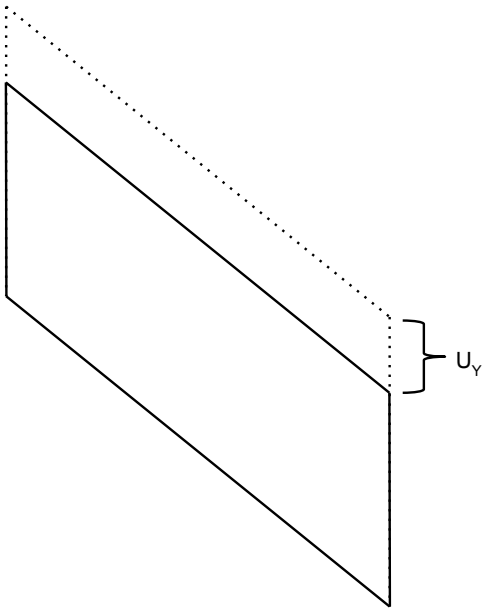


Figure B.1: Parallelepiped shear stress calculation

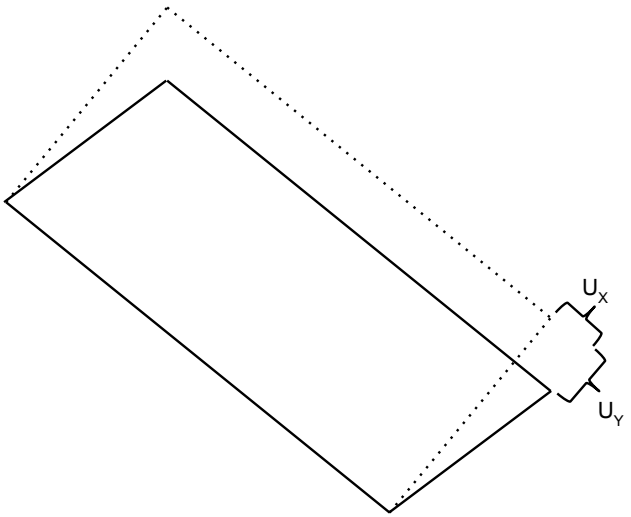


Figure B.2: Cube shear stress calculation

Appendix C

Energy Release Rate Calculation

In this project, the fractures are implemented such that they can propagate individually from each node along each axis. Referring to Figure C.1, where a node is located in the origin of the coordinate system, and there are 6 fracture vectors representing the fracture lenght along X,Y, and Z axis in both positive and negativ direction. Further when calculating the fracture propagation, we restrict the fractures to propagate between the finite elements i.e, in this case fractures can grow along the following planes: XZ, YX, and YZ. Also since we have different normal and shear stress at these three planes, we need to calculate the fracture propagation for the following case:

1. a_1 increase w.r.t XZ plane
2. a_5 increase w.r.t XZ plane
3. a_1 increase w.r.t YX plane
4. a_3 increase w.r.t YX plane
5. a_5 increase w.r.t YZ plane
6. a_3 increase w.r.t YZ plane

This would be the 6 cases for the finite element that exists in Figure C.1 where X,Y, and Z are positive. The neighbouring finite elements will then do the calculations

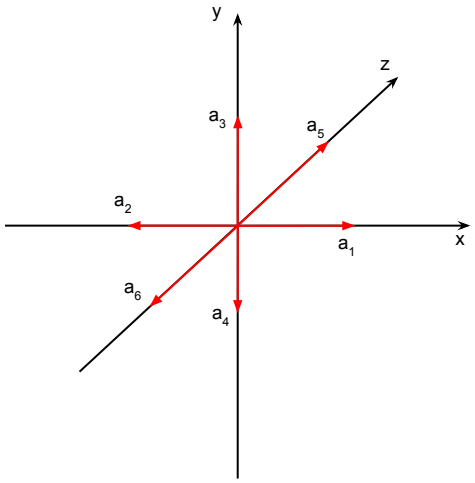


Figure C.1: Fracture propagation along axis

on the remaining fractures. We will then have to calculate the energy release rate for all these cases, because e.g. fracture a_1 may not propagate w.r.t YX plane, due to the stesses present at this plane, but it could propagate w.r.t the XZ plane due to the stresses in this plane. Below we will derive the equation for the energy release rate for case number 1:

$$\mathcal{G} = -\frac{d\Pi}{dA}$$

$$= -\frac{d\left(\Pi_0 - \frac{\sigma^2}{E}V\right)}{dA}$$

$$= -\frac{\left(\Pi_0 - \frac{\sigma^2}{E}V_e\right) - \left(\Pi_0 - \frac{\sigma^2}{E}V_i\right)}{A_e - A_i}$$

$$= -\frac{\frac{\sigma^2}{E}(V_i - V_e)}{A_e - A_i}$$

$$= -\frac{\frac{(N_3+S_5+S_1)^2}{E}\left(\left(\frac{||(\beta||a_1+a_5||)*(a_1\times a_5)||}{6}\right) - \left(\frac{||(\beta||(a_1+da)+a_5||)*((a_1+da)\times a_5)||}{6}\right)\right)}{\left(\frac{||(a_1+da)\times a_5||}{2}\right) - \left(\frac{||a_1\times a_5||}{2}\right)}$$

where N_3 is the normal stress along positive Y-axis, S_5 is the shear stress along positive Z-axis, and S_1 is the shear stress along positive X-axis (stress calculation is described in Section 4.1.4)

Appendix D

Movie

A short movie presentation of the results shown in this thesis has also been made, and can be viewed on the following web page:

`https://www.youtube.com/watch?v=xM5i0g6g8JM`

Appendix E

Detailed Simulation Results

In this chapter, you will find detailed simulation results in a form of a series of screenshots.

E.1 Energy Ratio for Parabola Terrain

In this section you can see the detailed simulation results from simulating homogeneous snow with a parabola terrain. And as we can see, the energy ratio builds up towards a value equal to one at the bottom of both sides, i.e. a fracture is propagating a distance $da = 10^{-11}$. But after this single propagation, the snow then stabilizes. The results are shown in Figure E.1 to E.11 (Page 125 - 130).

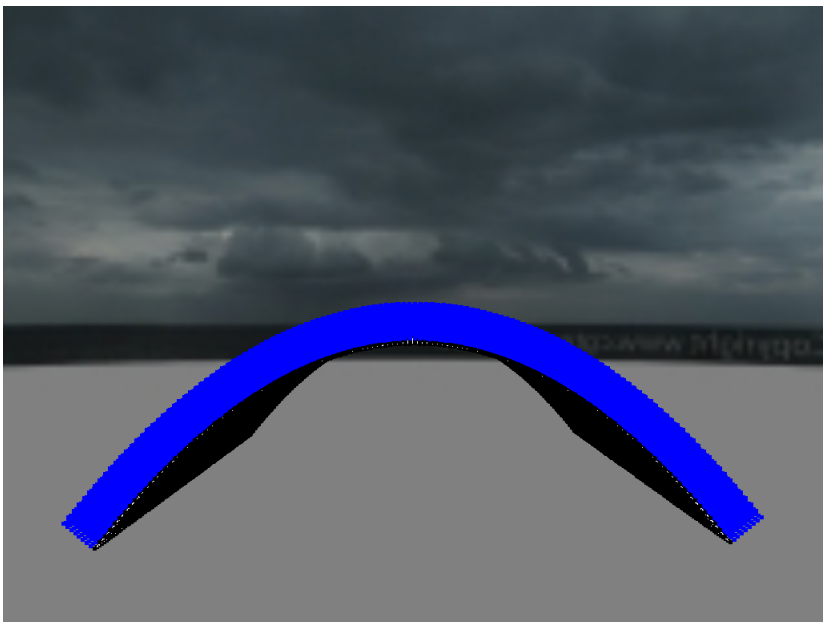


Figure E.1: Energy ratio for parabola terrain, step 1

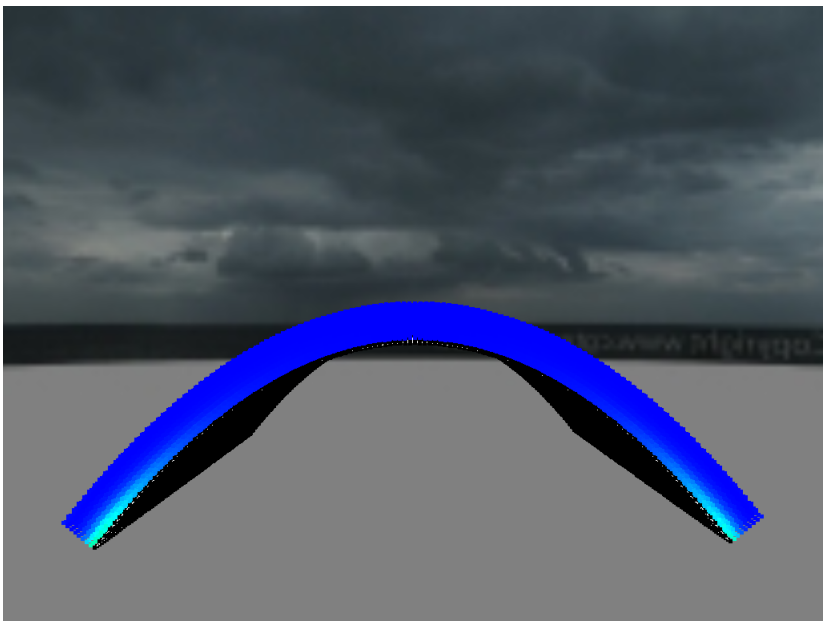


Figure E.2: Energy ratio for parabola terrain, step 2

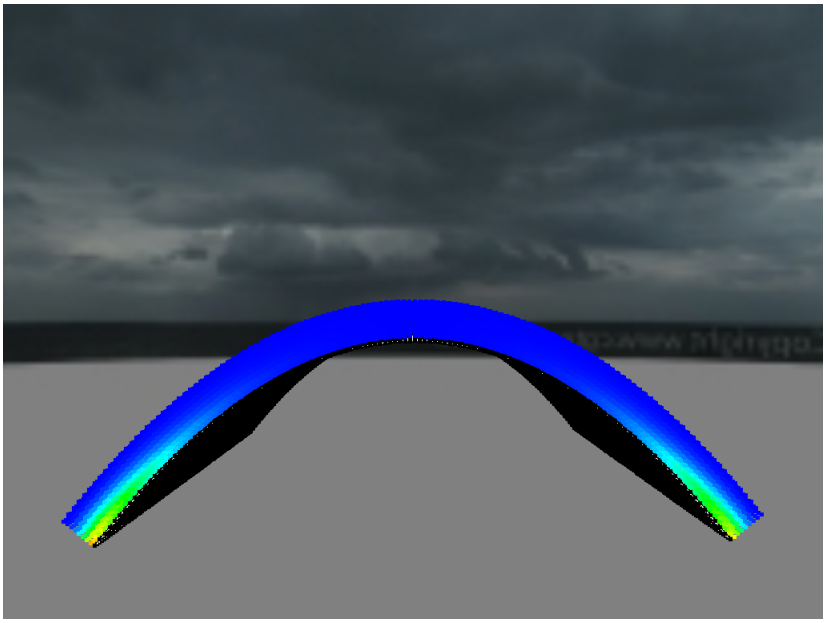


Figure E.3: Energy ratio for parabola terrain, step 3

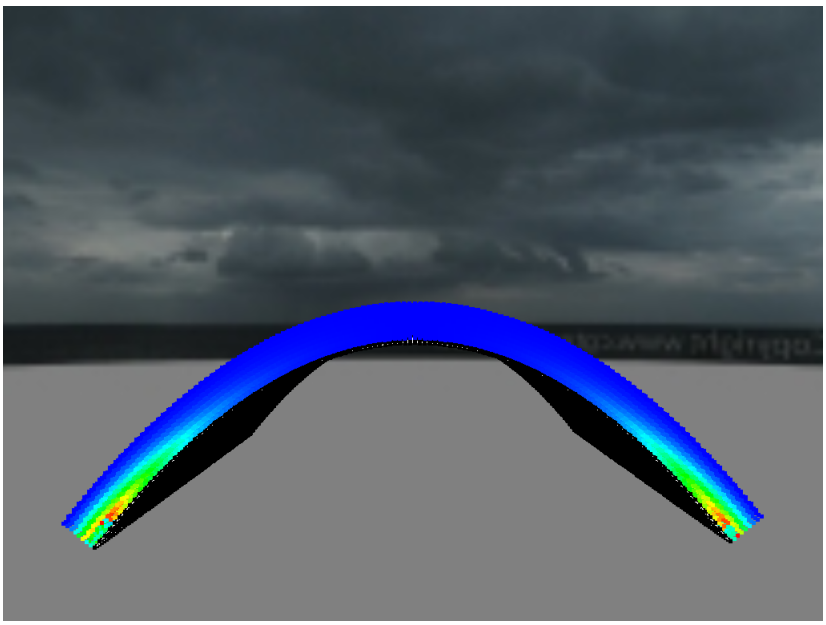


Figure E.4: Energy ratio for parabola terrain, step 4

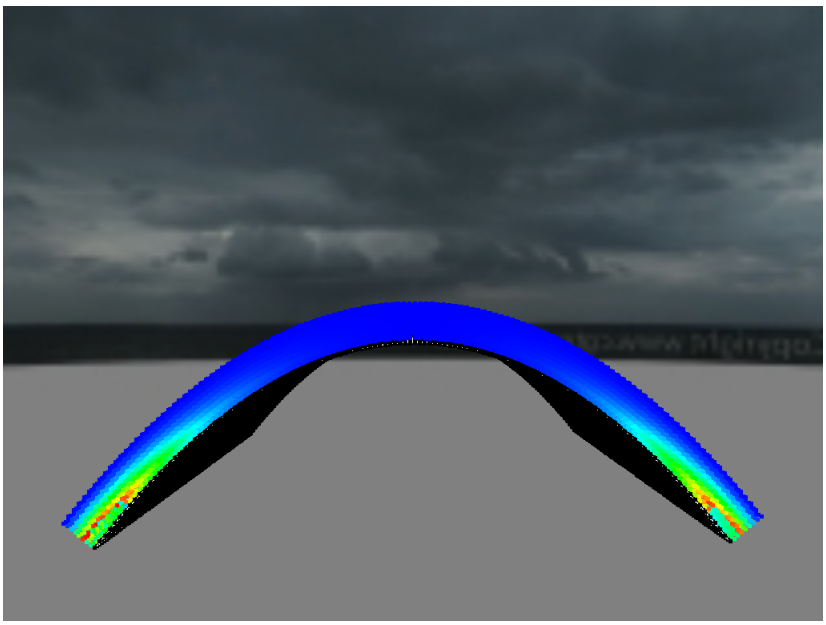


Figure E.5: Energy ratio for parabola terrain, step 5

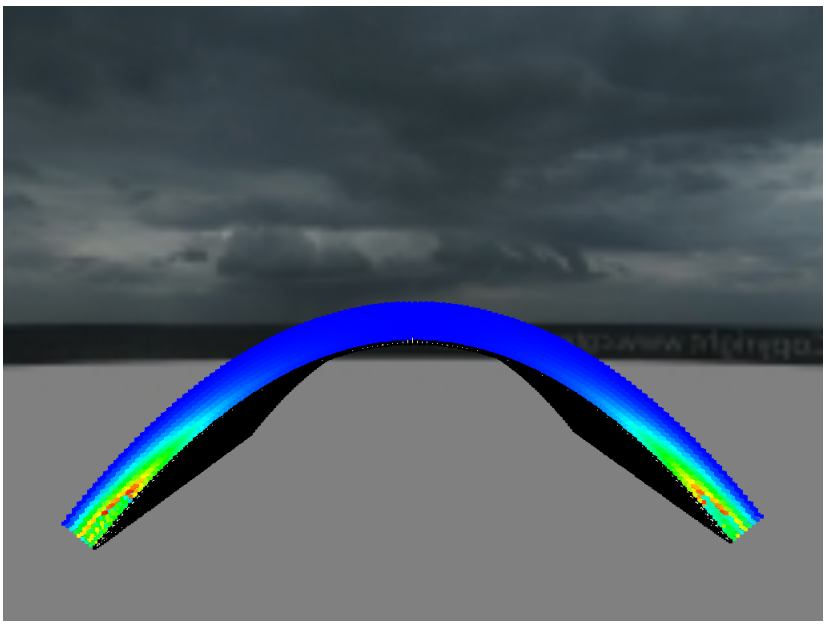


Figure E.6: Energy ratio for parabola terrain, step 6

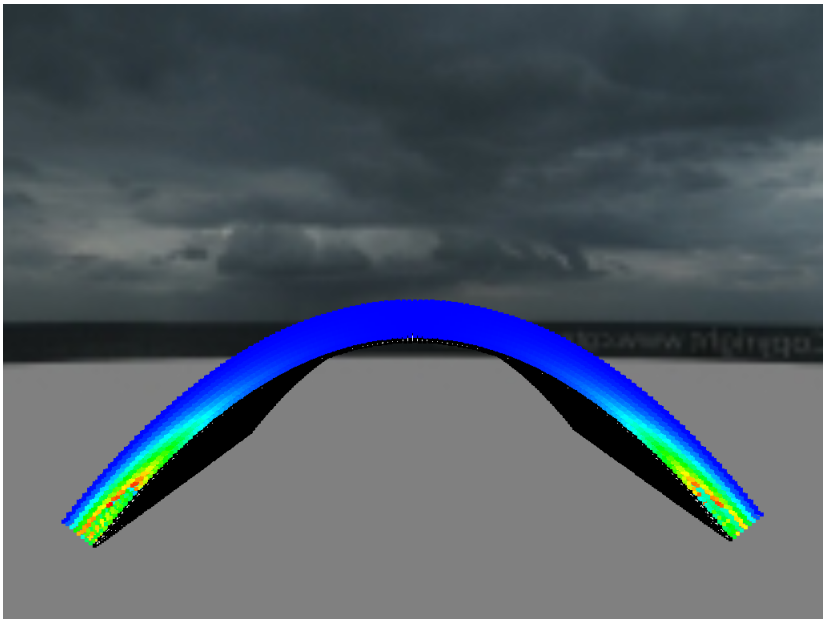


Figure E.7: Energy ratio for parabola terrain, step 7

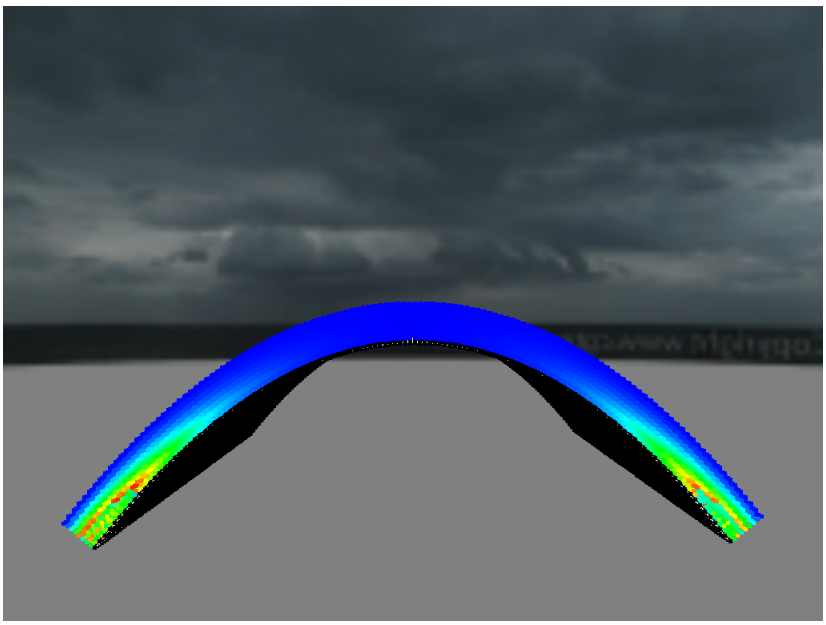


Figure E.8: Energy ratio for parabola terrain, step 8

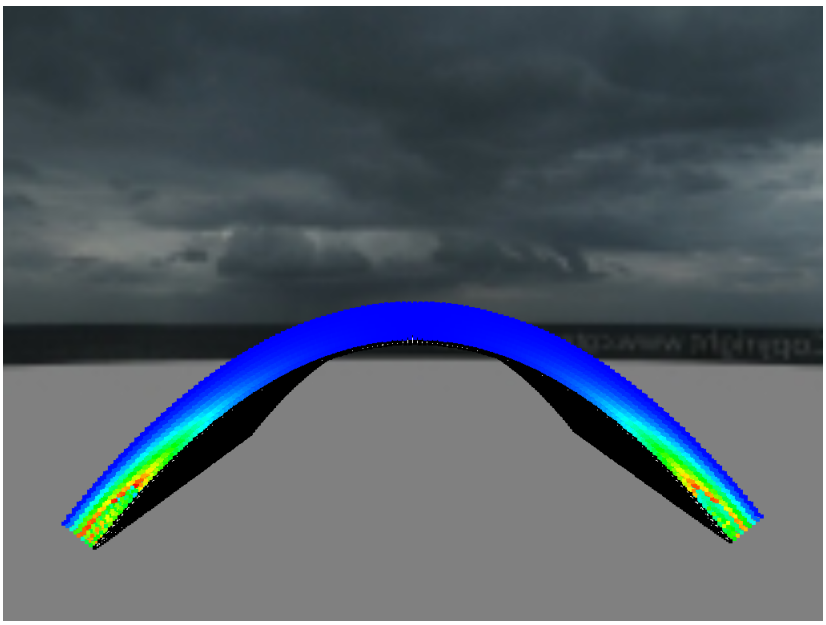


Figure E.9: Energy ratio for parabola terrain, step 9

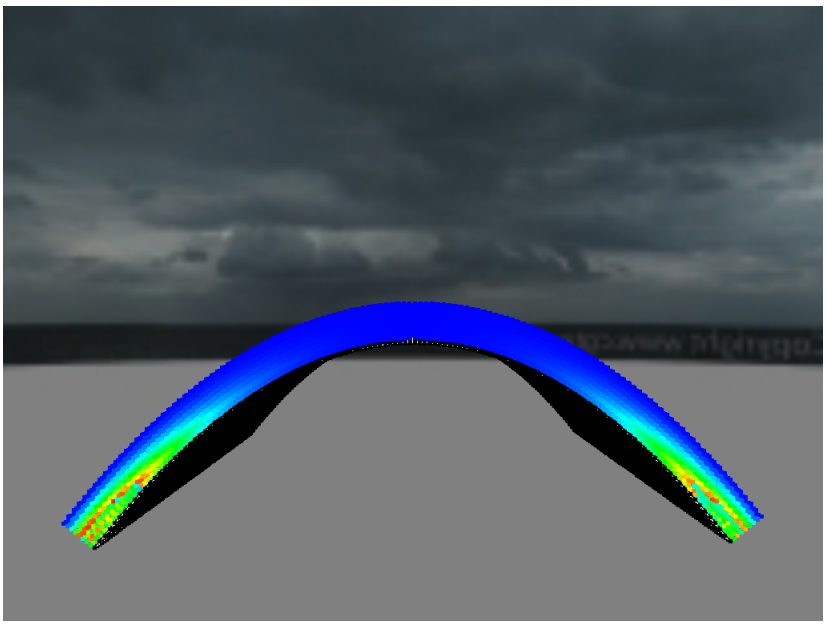


Figure E.10: Energy ratio for parabola terrain, step 10

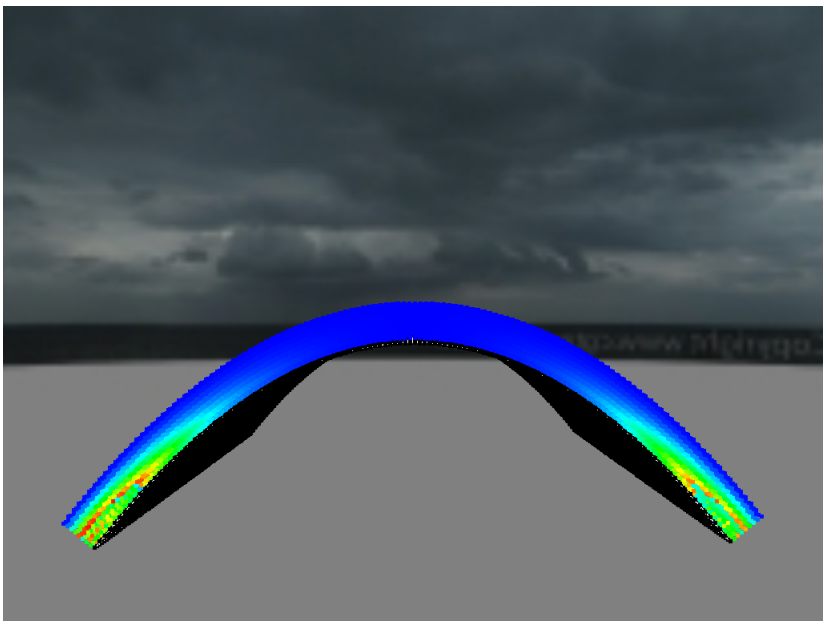


Figure E.11: Energy ratio for parabola terrain, step 11

E.2 Energy Ratio for Steep Slope Terrain

In this section you can see the detailed simulation results from simulating homogeneous snow with a steep slope terrain. The results are shown in Figure E.12 to E.23 (Page 131 - 136). And as we can see, the energy ratio is building up towards 1 at the center of the slope. Then suddenly a fracture process occurs, which immediately releases the stress, and the energy left in the structure is not enough to extend the fracture any further.

Figure E.12: Energy ratio for steep slope terrain, step 1

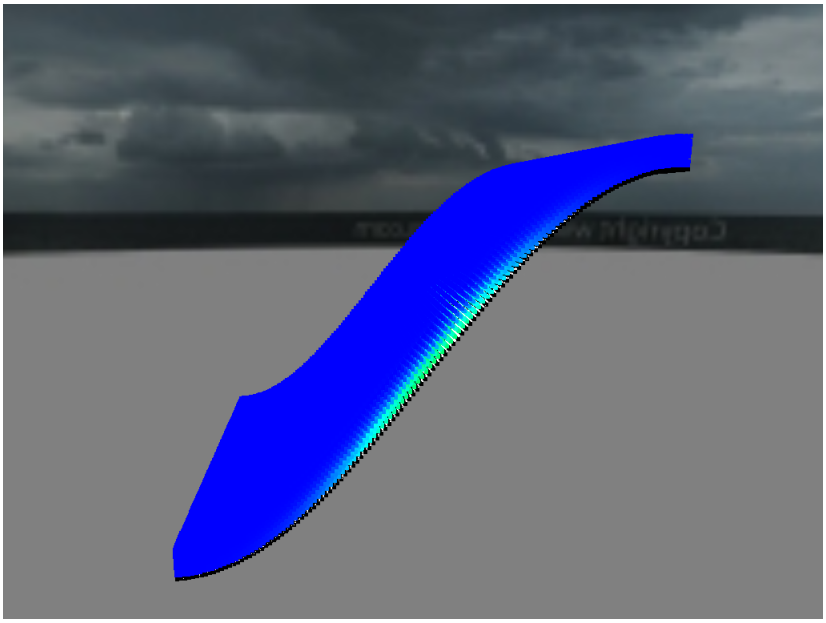


Figure E.13: Energy ratio for steep slope terrain, step 2

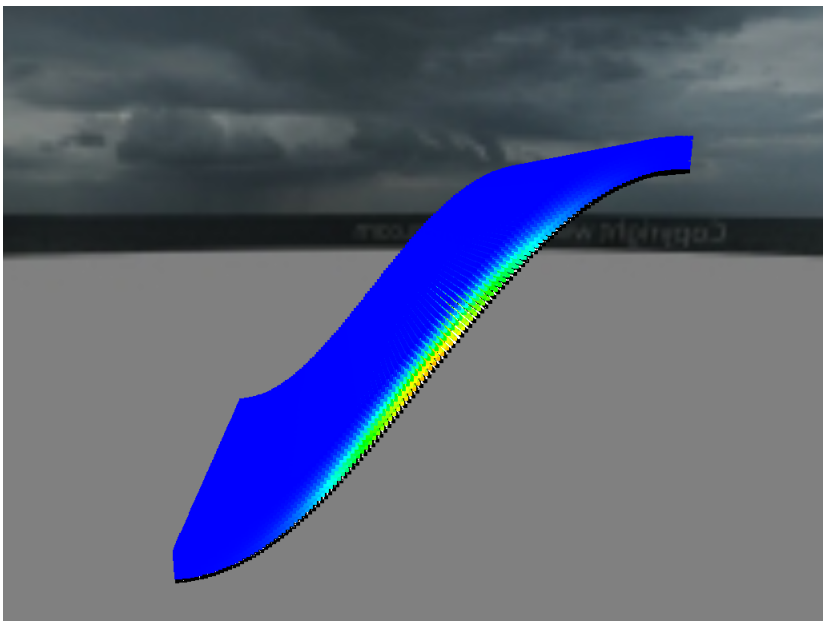


Figure E.14: Energy ratio for steep slope terrain, step 3

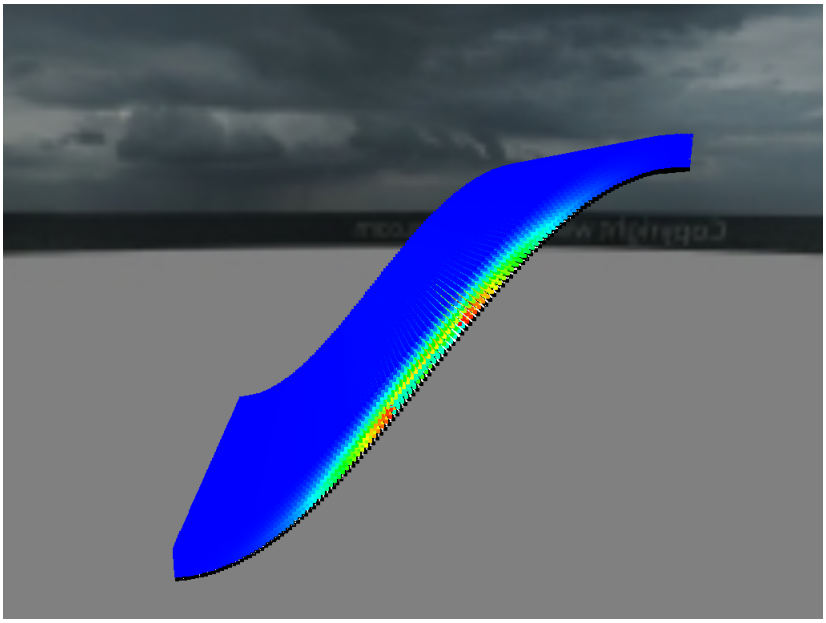


Figure E.15: Energy ratio for steep slope terrain, step 4

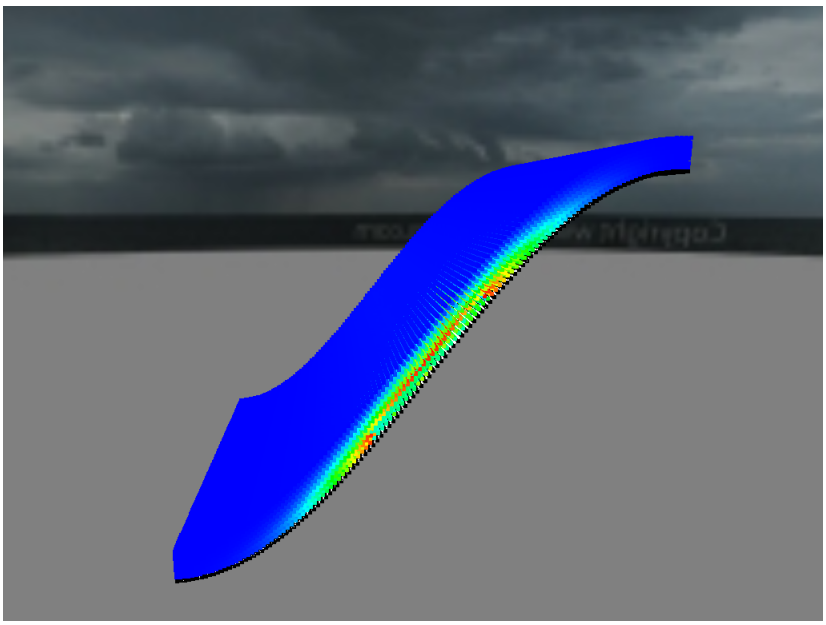


Figure E.16: Energy ratio for steep slope terrain, step 5

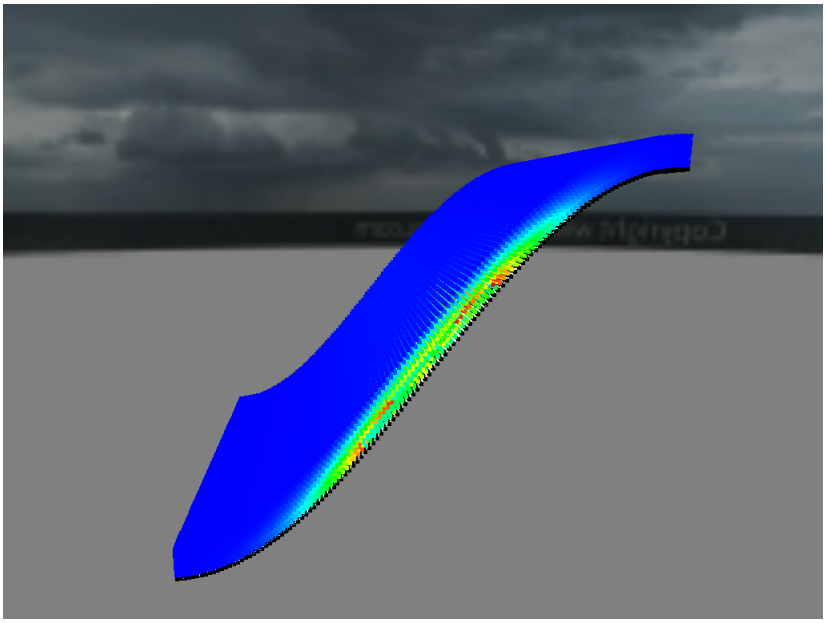


Figure E.17: Energy ratio for steep slope terrain, step 6

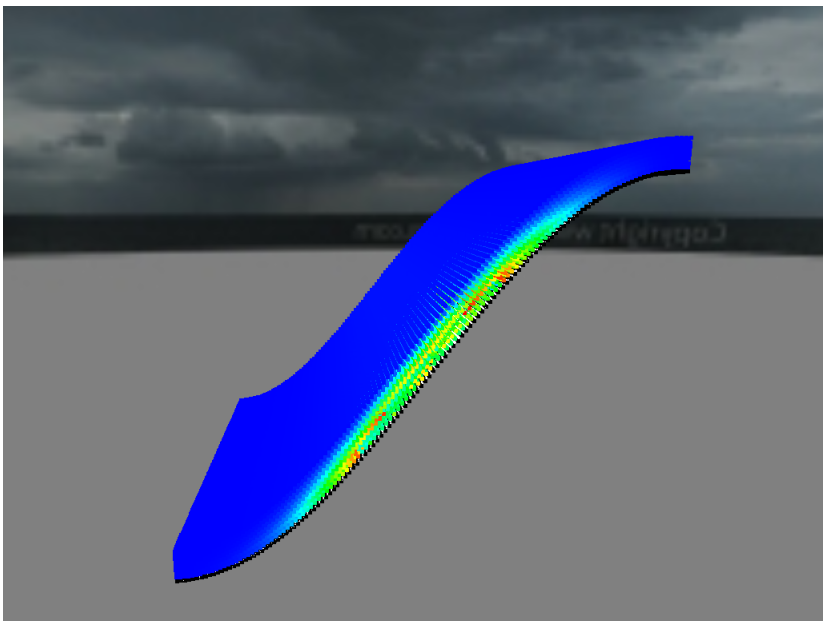


Figure E.18: Energy ratio for steep slope terrain, step 7

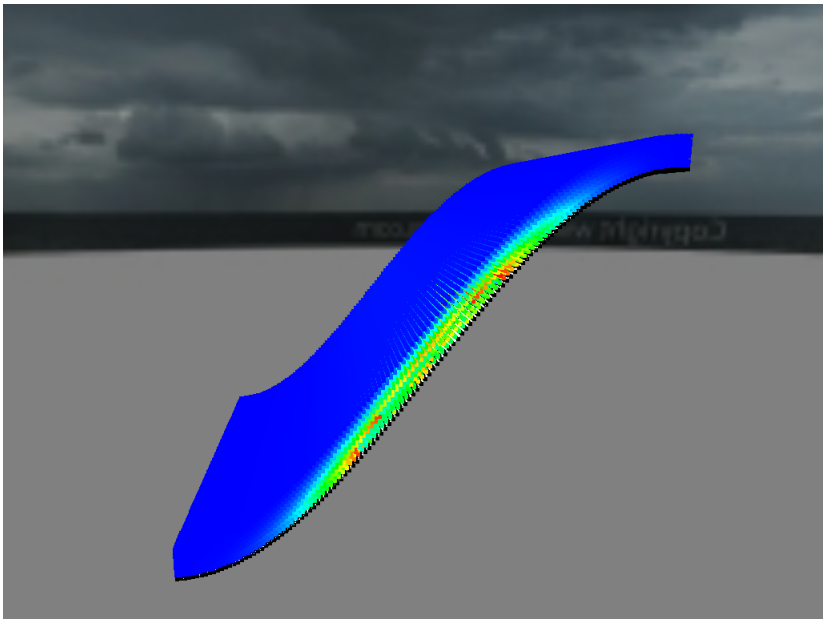


Figure E.19: Energy ratio for steep slope terrain, step 8

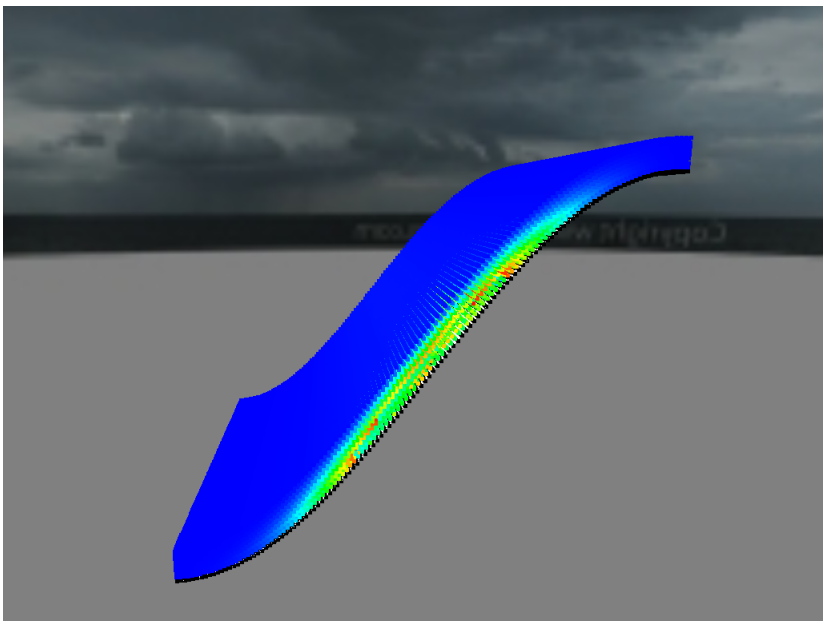


Figure E.20: Energy ratio for steep slope terrain, step 9

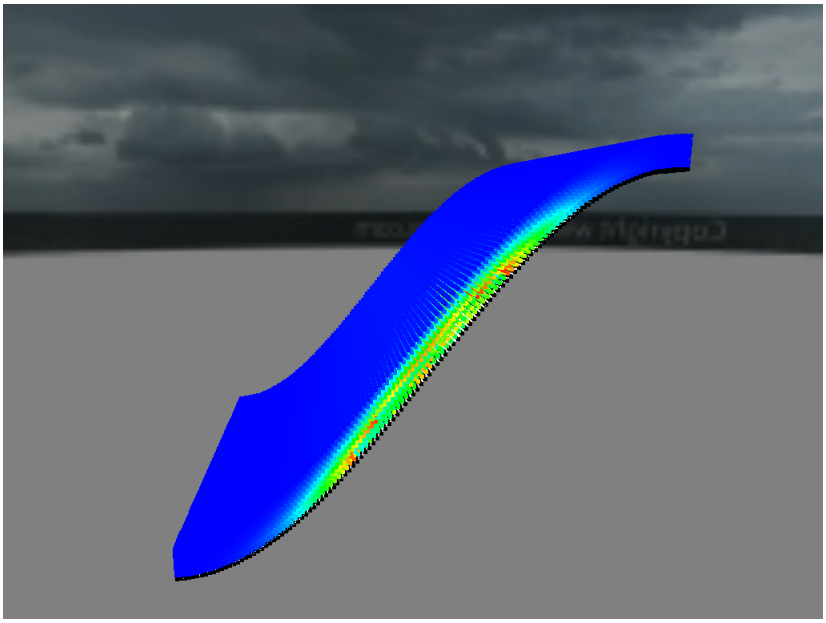


Figure E.21: Energy ratio for steep slope terrain, step 10

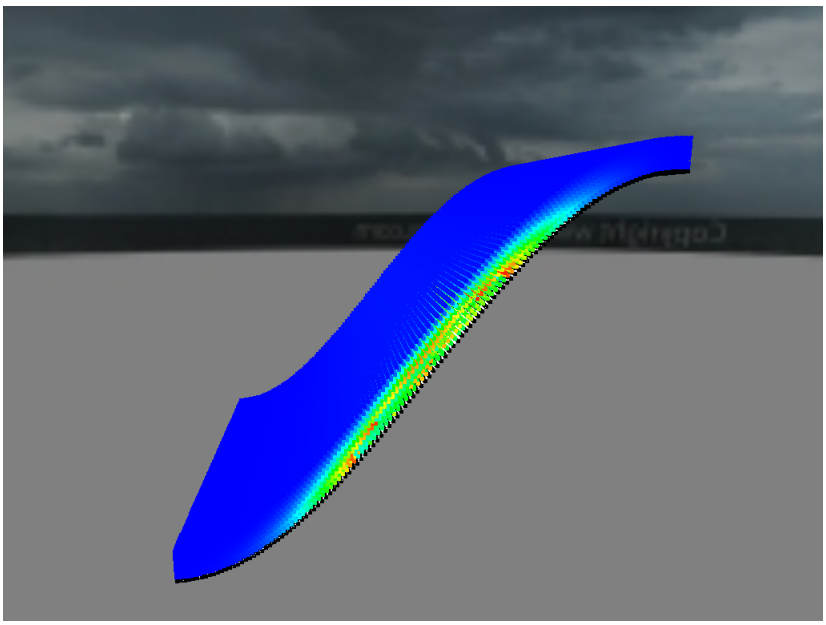


Figure E.22: Energy ratio for steep slope terrain, step 11

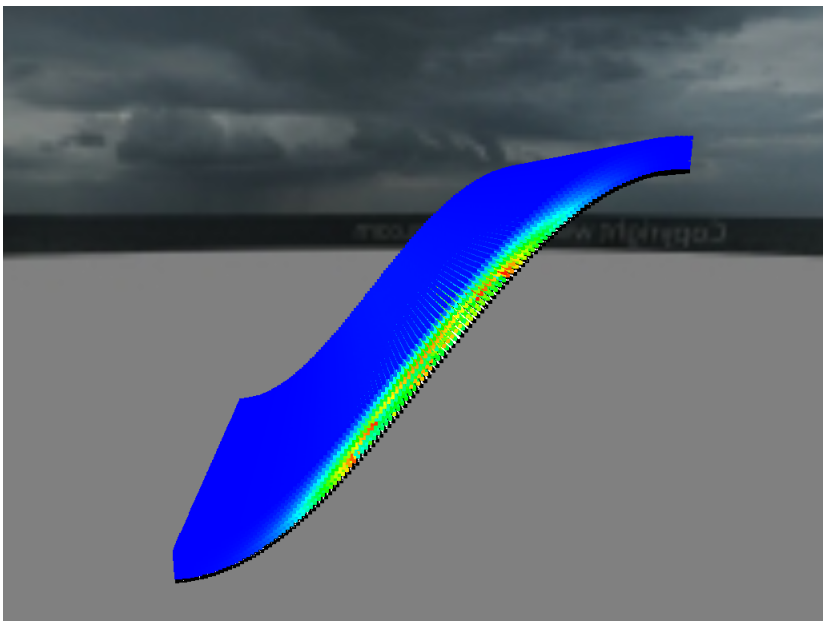


Figure E.23: Energy ratio for steep slope terrain, step 12

E.3 Timing Penalty of Double Precision

When calculating the time penalty of using double precision, detailed simulation timing was performned with different mesh sizes, where you can see the results in detail in Table E.1 on page 137.

Table E.1: Double versus single precision

Elements	Precision	GTX-480	C2070	GTX 760	K40c
576	Single	22.35 s	28.30 s	26.12 s	19.84 s
	Double	22.37 s	28.47 s	26.30 s	20.05 s
	Ratio	0.999	0.994	0.993	0.990
7 203	Single	22.97 s	29.28 s	27.93 s	20.58 s
	Double	23.49 s	29.67 s	28.59 s	20.82 s
	Ratio	0.978	0.987	0.977	0.988
21 904	Single	24.28 s	30.79 s	30.54 s	22.40 s
	Double	25.22 s	31.71 s	31.95 s	22.72 s
	Ratio	0.963	0.971	0.956	0.986
58 806	Single	26.69 s	33.62 s	35.73 s	25.96 s
	Double	29.31 s	36.04 s	39.51 s	26.68 s
	Ratio	0.910	0.933	0.904	0.973
107 632	Single	29.59 s	37.53 s	42.14 s	30.18 s
	Double	33.48 s	41.32 s	47.80 s	31.14 s
	Ratio	0.884	0.908	0.882	0.969
199 809	Single	34.70 s	45.05 s	54.47 s	38.52 s
	Double	43.92 s	53.66 s	67.42 s	41.35 s
	Ratio	0.790	0.840	0.808	0.932
302 760	Single	41.37 s	52.97 s	68.31 s	47.69 s
	Double	53.04 s	62.70 s	84.14 s	51.24 s
	Ratio	0.780	0.845	0.812	0.931
475 212	Single	50.14 s	64.41 s	90.01 s	62.27 s
	Double	63.82 s	77.23 s	111.01 s	64.93 s
	Ratio	0.786	0.834	0.811	0.959
702 464	Single	62.44 s	80.67 s	120.24 s	82.20 s
	Double	83.60 s	101.9 s	156.00 s	87.92 s
	Ratio	0.747	0.791	0.771	0.935
930 015	Single	73.98 s	96.91 s	150.22 s	101.95 s
	Double	100.23 s	125.10 s	192.96 s	107.14 s
	Ratio	0.738	0.775	0.779	0.952
1 276 292	Single	96.37 s	127.69 s	196.61 s	132.46 s
	Double	141.40 s	167.40 s	263.53 s	142.33 s
	Ratio	0.682	0.763	0.746	0.931
1 609 218	Single	110.95 s	142.83 s	234.06 s	156.76 s
	Double	174.96 s	193.22 s	313.49 s	172.01 s
	Ratio	0.634	0.739	0.747	0.911

Appendix F

Poster

A poster has also been created for this project and is shown on page 139.

Avalanche Prediction using Fracture Mechanics

Fracture Mechanics

- Using the "Energy Release Rate" to determine fracture propagation

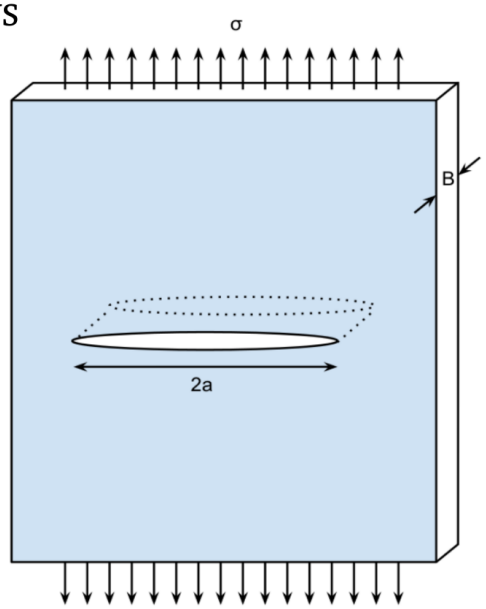
$$\mathcal{G} = -\frac{d\Pi}{dA} = -\frac{d\left(\Pi_0 - \frac{\pi\sigma^2 a^2 B}{E}\right)}{dA}$$

- Fractures can propagate when the energy release rate reaches the critical energy release rate:

$$\mathcal{G}_c = \frac{dW_s}{dA}$$

Fracture Mechanics

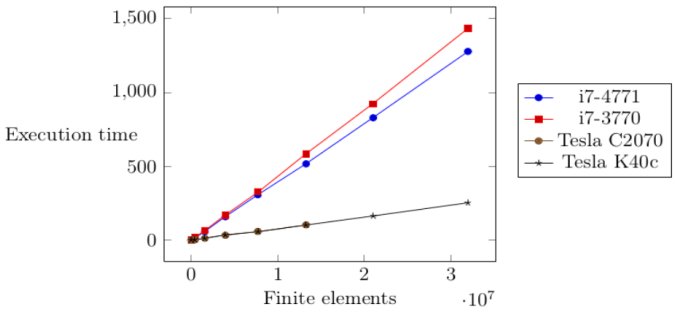
Fractures can more easily propagate when material are subjected to microflaws



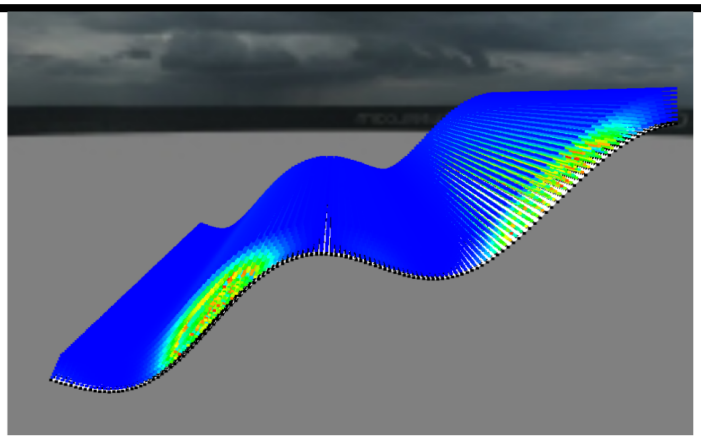
GPU Utilization

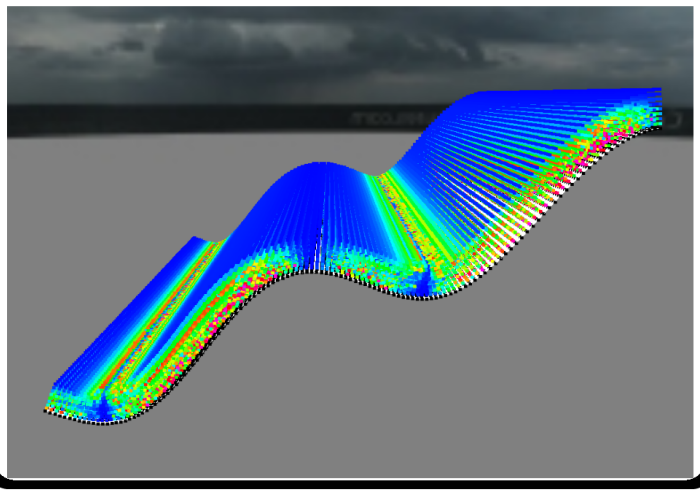
- 5x speedup compared to parallel CPU execution

- Additional gain for Kepler GPU if shared memory is utilized



Results





HPC Research Group

NTNU – Trondheim
Norwegian University of Science and Technology

Appendix G

First Approach

The implementation of this project was not the original approach that I had in mind. After I had learnt enough about fracture mechanics I was very eager to start on the project and obtain some results. So I started quite early on with another approach to solve the problem of this thesis, and also wrote a lot of this method while I was implementing. I also had very little knowledge about the finite element method when I wrote and implemented the first approach, and it was only after much time had been spent on the implementation that I found out that I had to take a step back, and obtain more knowledge on the finite element method before I started the actual implementation.

Elemental Crack Advance

In this project the **Elemental crack advance** will be implemented in order to calculate fracture propagation in the snow layers. The external load on each snow layers will be the above snow masses, and the stresses in each snow layer will be calculated from this load. The energy release rate $\mathcal{G}$ will then be calculated from equation 2.15, which requires knowledge of the stress field and the crack area a. We will also assume the existence of micro-cracks in the snow layers, which is a valid assumptions due to the ice-matrix. However, how big these micro-cracks should be needs to be figured out, and the size of these existing micro-cracks should also depend on the snow-layer properties.

The method of choice for solving the fracture problem is based on the energy release rate, which is based on calculating the derivative of the potential energy of the structure w.r.t. the crack-area. However, this method requires a low dA to be accurate. The calculations are done in the following manner, where the energy release rate is calculated, and is then compared to the fracture toughness $\mathcal{G}_c$. And finally when the energy release rate is equal to the fracture toughness, the crack can propagate a given distance da:

$$\Pi\left(a\right) = \Pi_0 - \frac{\pi\sigma^2 a^2 B}{E} \tag{G.1}$$

where Π is the potential energy of the structure, Π_0 is the potential energy of an uncracked structure, σ is the stress field, a is the crack lenght, B is the thickness of the structure (Figure 2.8), and E is young's modulus.

$$\mathcal{G} = -\frac{d\Pi\left(a\right)}{dA} = -\frac{\Pi(a+\Delta a) - \Pi(a)}{dA} \tag{G.2}$$

Then after the energy release rate is calculated, we can compare it with $\mathcal{G}_c$.

$$\mathcal{G}_c = \frac{dW_s}{dA} = \frac{d\left(4aB\left(\gamma_s + \gamma_p\right)\right)}{dA} = \frac{d\left(4aBw_f\right)}{dA} = 2w_f \tag{G.3}$$

where $w_f = \gamma_s$ for ideally brittle materials, or $w_f = \gamma_s + \gamma_p$ for materials which experiences plastic flow at the crack-tip.

FEM Implementation

To calculate the fracture process, we will have to calculate the derivative of the energy release rate w.r.t. the change of the fracture area, and we will need to solve Equation G.2 to calculate this. This equation relies on the stress field in the structure, the size of the fractures, and Young's modulus which is a material constant. The calculation of the stress field is described in Section G, but we need a representation of the actual fracture propagation. To represent the fractures, we will restrict fractures to propagate along the interface between the finite elements. How far the fractures have propagated will be represented by 8 float values for each node. Where each float value will represent how far the fracture has propagated in $\{+x, -x, +y, -y, +z, -z, +d, -d\}$ directions, where $\pm d$ is the fracture propagation along the diagonal for the bottom surface.

The reason why we have factures on the diagonal on the bottom surface is due to the terrain heightmap, and how the finite elements are constructed. The terrain heightmap is a two dimensional array with height values at each $\{x, z\}$ coordinate, the finite elements are then constructed by a number of nodes above each terrain height with a specified dy. This gives us that the bottom and top surface plane represented by the 4 bottom and top nodes respectively will not necessarily be located in the same plane, however the 4 vertices on the remaining 4 sides will be located in the same plane.

The finite element construction is displayed in Figure G.3, where the solid lines represent the terrain and the dashed lines represent a element, and here we can see clearly that the two triangles representing the bottom and top surface of each element will not necessarily be located in the same plane, but the remaining sides will be located in the same plane.

Triaxial Stress Calculation

Calculating the stress field in the entire structure in the most important calculation, and this stress field will be represented by two 3D vectors for each interface between two finite elements. One vector representing the normal stress to the neighbouring element, and one vector representing the shear stress to the neighbouring element.

Figure G.6 shows the forces which will act on the neighbouring elements, when the external force is gravity only. For calculating the normal and shear stress acting on the plane separating the current element and the neighbouring elements, we will need to decompose the gravity vector $\vec{g}$ into two vectors which are perpendicular and parallel to the bottom surface. These vectors are represented as g_2 and g_1 respectively. We can then calculate the normal stress and the shear stress acting on the bottom surface and all the 4 sides.

Firstly we need to calculate the normal vector on the bottom surface, but since the bottom surface consist of two separate triangles we will repeat the below calculations for both triangles. To calculate the stress acting on each surface, we

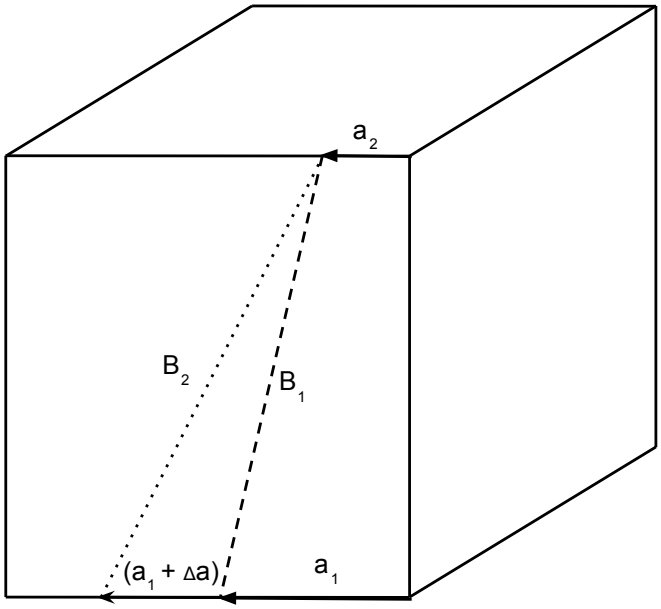


Figure G.1: Representation of crack propagation along sides on a single element

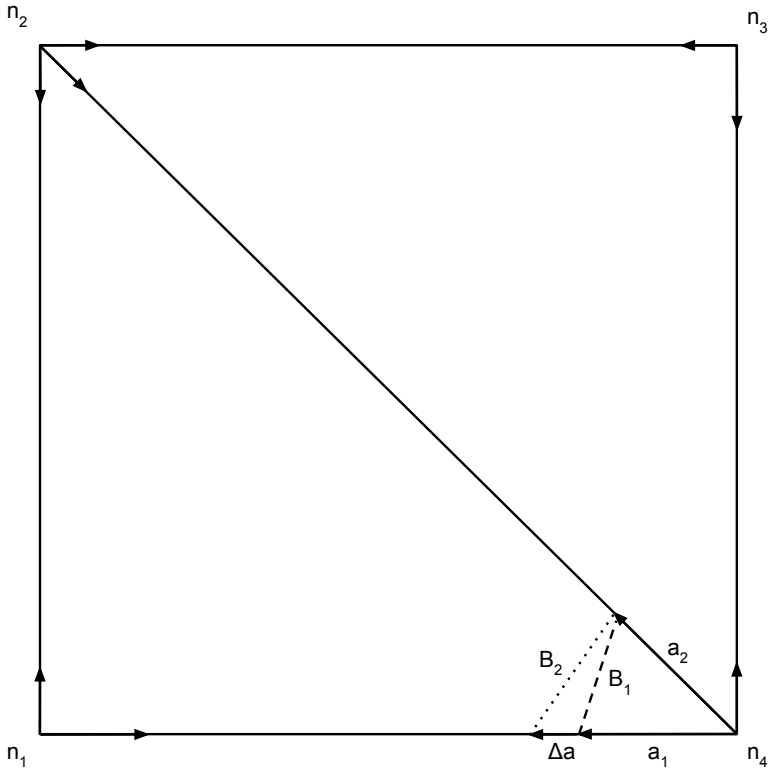


Figure G.2: Representation of crack propagation along bottom surface on a single element

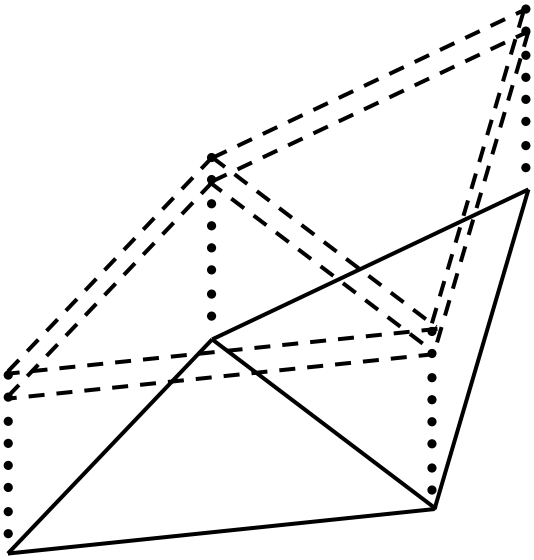


Figure G.3: Element nodes over terrain heightmap

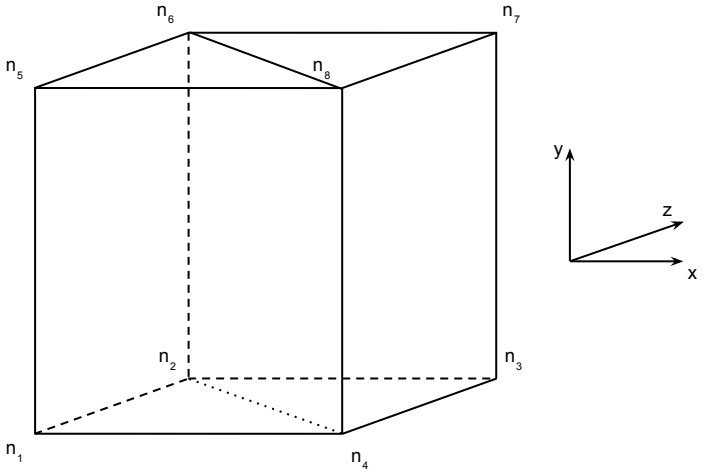


Figure G.4: Finite element vertex indexing

will have to decompose the gravity vector as displayed in Figure G.6. To calculate these decomposed vectors, we first have to calculate the normal vector for each triangle on the bottom surface by the following:

$$\vec{n_1} = \vec{n_{14}} \times \vec{n_{12}} \tag{G.4}$$

$$\vec{n_2} = \vec{n_{32}} \times \vec{n_{34}} \tag{G.5}$$

where $\vec{n_{xy}}$ denotes the vector from point x to point y, incices are displayed in Figure G.4

Then we need to calculate the angle of the terrain slope (remaining calculations are performed for both normal vectors), and this is easily calculated by taking advantage of a spherical coordinate system. In a spherical coordinate system, a vector can be represented by a radius, an angle $\theta \exists [0, \pi]$, and a angle $\phi \exists [0, 2\pi]$ as displayed in Figure G.5[1]. Then to calculate the angle of the terrain, we can use

[1]Public domain, reprinted from `http://en.wikipedia.org/wiki/File:3D_Spherical.svg`

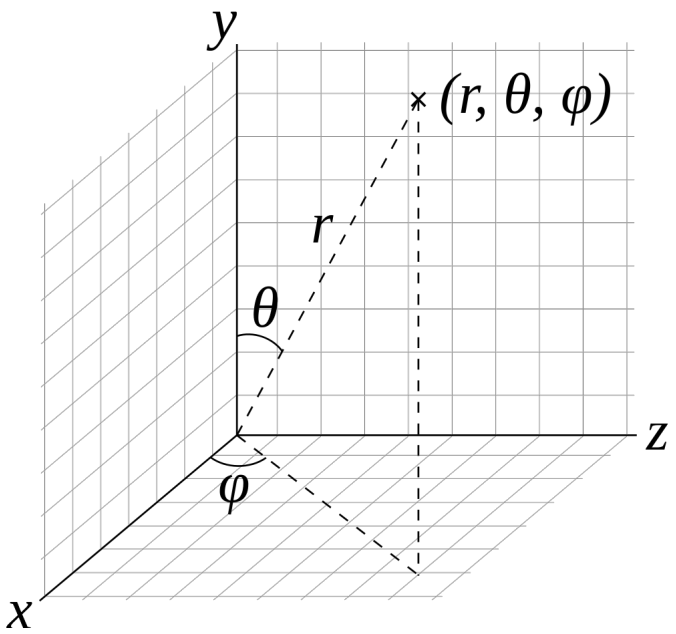


Figure G.5: Spherical coordinate system

the following equation:

$$\theta = \frac{\pi}{2} - acos\left(\frac{n_y}{||n||}\right) \tag{G.6}$$

where $||n||$ denotes the euclidean norm

$$\varphi = atan\left(\frac{n_z}{n_x}\right) \tag{G.7}$$

We can then use the following equations to find the spherical representation of vector g_1 and g_2, then afterwards we can convert them to cartesian coordinates:

$$g_{2r} = m \times 9.81 \times cos\,(\theta)$$

$$g_{2\theta} = \tfrac{\pi}{2} + \left(\tfrac{\pi}{2} - \theta\right) = \pi - \theta$$

$$g_{2\varphi} = \varphi$$

$$g_{1r} = m \times 9.81 \times sin\,(\theta)$$

$$g_{1\theta} = \tfrac{\pi}{2} + \left(\tfrac{\pi}{2} - \left(\tfrac{\pi}{2} - \theta\right)\right) = \tfrac{\pi}{2} + \theta$$

$$g_{1\varphi} = \varphi$$

where g_{xy} is the property y of the x'th decomposed vector of g, and $y \ni \{r, \theta, \varphi\}$, and m is the above snow masses

We can then finally convert these values into cartesian coordinates by the following equations:

$$g_1 = \begin{bmatrix} g_{1r} \times sin(g_{1\theta}) \times cos(g_{1\varphi}) \\ g_{1r} \times cos(g_{1\theta}) \\ g_{1r} \times sin(g_{1\theta}) \times sin(g_{1\varphi}) \end{bmatrix}$$

$$g_2 = \begin{bmatrix} g_{2r} \times sin(g_{2\theta}) \times cos(g_{2\varphi}) \\ g_{2r} \times cos(g_{2\theta}) \\ g_{2r} \times sin(g_{2\theta}) \times sin(g_{2\varphi}) \end{bmatrix}$$

Table G.1: Forces acting on side planes

Force	XY-plane	ZY-plane
Normal	x-component	z-component
Shear 1	z-component	x-component
Shear 2	y-component	y-component

We now have all the vectors we need to figure out normal and shear stress for the bottom surface, and all 4 sides. For the bottom surface, g_1 and g_2 represent the normal and shear stress, and for each side we have all the necessary data stored in the g_1 vector.

If we look at Figure G.5, and imagine that the calculated g_1 vector which is parallel to the slope is the vector from (r, θ, φ) to $(0, 0, 0)$. Then we will have the following components of the g_1 vector representing the normal and shear forces acting on these sides as shown in Table G.1, the other g_1 vector calculated for the opposite triangle will then be used for the other two sides of the element. More accurate, if we look at Figure G.4, we will have a g_{11} vector representing the forces acting on the surfaces represented by $\{n_1, n_2, n_6, n_5\}$ and $\{n_1, n_4, n_8, n_5\}$, and another vector g_{12} representing the forces acting on the surfaces represented by $\{n_3, n_2, n_6, n_7\}$ and $\{n_3, n_4, n_8, n_7\}$. And g_{11} and g_{12} is the decomposed gravity vector acting parallel to the bottom surface triangles $\{n_1, n_2, n_4\}$ and $\{n_3, n_2, n_4\}$ respectively.

The next step in calculating the stress acting on each surface is the area of them, this is easily performed by using 2D vectors and calculating the determinants for each side of the element, and by using half of the length of the cross-product vector for the bottom surface triangles. The resulting surface area equations are therefore:

$$A\left(\{n_1, n_2, n_6, n_5\}\right) = |det\left(\begin{bmatrix} n_{5y} - n_{1y} & n_{5z} - n_{1z} \\ n_{2y} - n_{1y} & n_{2z} - n_{1z} \end{bmatrix}\right)|$$

$$A\left(\{n_1, n_4, n_8, n_5\}\right) = |det\left(\begin{bmatrix} n_{5y} - n_{1y} & n_{5x} - n_{1x} \\ n_{4y} - n_{1y} & n_{4x} - n_{1x} \end{bmatrix}\right)|$$

$$A\left(\{n_3, n_2, n_6, n_7\}\right) = |det\left(\begin{bmatrix} n_{7y} - n_{3y} & n_{7x} - n_{3x} \\ n_{4y} - n_{3y} & n_{4x} - n_{3x} \end{bmatrix}\right)|$$

$$A\left(\{n_3, n_4, n_8, n_7\}\right) = |det\left(\begin{bmatrix} n_{7y} - n_{3y} & n_{7z} - n_{3z} \\ n_{4y} - n_{3y} & n_{4z} - n_{3z} \end{bmatrix}\right)|$$

$$A\left(\{n_1, n_2, n_4\}\right) = \tfrac{1}{2} length\left(\vec{n_{12}} \times \vec{n_{14}}\right)$$

$$A\left(\{n_3, n_4, n_2\}\right) = \tfrac{1}{2} length\left(\vec{n_{32}} \times \vec{n_{34}}\right)$$

where *det* denotes the determinant of a vector in $\mathbb{R}^2$, and *length* denotes the euclidean norm in $\mathbb{R}^3$.

Energy Release Rate Calculation

For calculating how the fracture propagates in the structures, the energy release rate $\mathcal{G}$ must be calculated (Equation G.2). In Section G we looked at how we

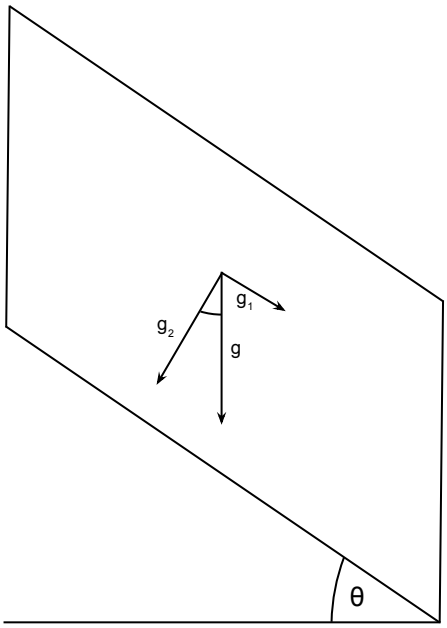


Figure G.6: Forces acting on neighbouring elements

will represent the fractures. We will now look at how we will calculate the energy release rate by a change Δa in fracture length:

$$\mathcal{G} = \frac{d\Pi}{dA}$$

$$= \frac{d\left(\Pi_0 - \frac{\pi\sigma^2 a^2 B}{E}\right)}{dA}$$

$$= \frac{\left(\Pi_0 - \frac{\pi\sigma^2||a_1+\Delta a||^2\cdot||B_2||}{E}\right) - \left(\Pi_0 - \frac{\pi\sigma^2||a_1||^2\cdot||B_1||}{E}\right)}{\frac{1}{2}||B_1 \times B_2||}$$

$$= \frac{\frac{\pi\sigma^2}{E}\left((||a_1||^2 \cdot ||B_1||) - (||a_1 + \Delta a||^2 \cdot ||B_2||)\right)}{\frac{1}{2}||B_1 \times B_2||}$$

Where a is the fracture vector, B_1 and B_2 is displayed in Figure G.1

Then after the energy release rate is equal to $\mathcal{G}_c$ the fracture can be extended a distance Δa. These calculations will then be performed for each node in the direction along the surface of the element. In order words, node n_1 will not try to extend the fracture in $-x$ direction, because this is outside of the element. But this node can extend a fracture in the following directions $\{+x, +z, +y\}$ (Figure G.4).

Memory Requirement

In this section we will look at the necessary data that we need to store in order to calculate the energy release rate, which will be compared to the fracture toughness $\mathcal{G}_c$.

$terrain_x$	$terrain_z$	max_snow	{dx, dy, dz}	Size (MiB)
128	128	4	{0.5, 0.05, 0.5}	80
128	128	4	{0.5, 0.01, 0.5}	400
768	768	4	{0.5, 0.05, 0.5}	2880

Table G.2: Memory requirement for calculating Young's modulus

$terrain_x$	$terrain_z$	max_snow	{dx, dy, dz}	Size (MiB)
128	128	4	{0.5, 0.05, 0.5}	20
128	128	4	{0.5, 0.01, 0.5}	100
768	768	4	{0.5, 0.05, 0.5}	720

Table G.3: Memory requirement for calculating crack area

At the start of this project, it quickly became clear that the memory requirements for storing various data would be huge. And the terrain size has been greatly reduced when compared to earlier avalanche prediction projects [26].

Stresses will be represented by two 3D vectors on each plane of the elements, which sums up to a total of:

$$\frac{terrain_x \times terrain_z \times max\_snow\_height \times 2 \times 3 \times 4}{dx \times dy \times dz} \tag{G.8}$$

Snow layer displacement should be modelled, and is relatively simple to calculate when the Young's modulus is known. Then to calculate the snow layer displacements would need to store the vertices of all snow layers, and update the Y-coordinate when the snow layers are compressed. In this thesis we will assume that the snow layers are only compressed, and that the original top surface area do not change. The memory requirement for storing all snow layer vertices would then be:

$$\frac{terrain_x \times terrain_z \times max\_snow\_height\,(3 \times 4 + 4)}{dx \times dy \times dz} \tag{G.9}$$

Where {x,y,z} coordinates are stored, and an additional debugging variable

Which gives memory requirement displayed in Table G.2 for various configurations. The reason why snow layer displacement should be modelled is because of this is most likely happening in reality in some degree, and this would also minimize the crack-lenght required to neighbouring snow-layers.

Crack area is also necessary to calculate the fracture toughness of a material, and this will be estimated by a single float value per mesh point. And will therefore have the following memory requirements:

$$\frac{terrain_x \times terrain_z \times max\_snow\_height \times 4}{dx \times dy \times dz} \tag{G.10}$$

Which gives us the memory requirements displayed in Table G.3 for various configuration.

Appendix H

Code

In this chapter we will list detailed code for different parts of the simulation.

H.1 Mesh Generation

Listing H.1: Bottom snow layer mesh generation

```
int count = 0;
int nx = (int)((resolution-1)/mesh_dx)+1;
int nz = (int)((resolution-1)/mesh_dz)+1;

float x,z;
z = 0.f;
for(int i = 0; i < nz; i++){
    x = 0.f;
    for(int j = 0; j < nx; j++){
        data[count].x = (SCENE_X*x/resolution);
        data[count].y = (float)vertices[(int)(((int)(z))*resolution+((int)(x)))].y
            ;
        data[count].z = (SCENE_Z*z/resolution);
        data[count].w = 0.f;

        count++;

        assert(count <= num_nodes);
        x += mesh_dx;
    }
    z += mesh_dz;
}
```

Listing H.2: Remaining snow layer mesh generation

```
// Creating snow layers, based on normal vectors from layer below
glm::vec3 v1, v2, v3, v4, n;
for(int y = 1; y < max_snow_layers; y++){
    for(int z = 0; z < nz; z++){
        for(int x = 0; x < nx; x++){
            // Creating v1 vector
            if(z > 0){
                v1.x = data[count - nx*nz - nx].x - data[count - nx*nz].x;
                v1.y = data[count - nx*nz - nx].y - data[count - nx*nz].y;
                v1.z = data[count - nx*nz - nx].z - data[count - nx*nz].z;
            }

            // Creating v2 vector
            if(x > 0){
                v2.x = data[count - nx*nz - 1].x - data[count - nx*nz].x;
                v2.y = data[count - nx*nz - 1].y - data[count - nx*nz].y;
                v2.z = data[count - nx*nz - 1].z - data[count - nx*nz].z;
            }

            // Creating v3 vector
            if(z < nz-1){
                v3.x = data[count - nx*nz + nx].x - data[count - nx*nz].x;
```

```
23                  v3.y = data[count - nx*nz + nx].y - data[count - nx*nz].y;
24                  v3.z = data[count - nx*nz + nx].z - data[count - nx*nz].z;
25              }
26
27              // Creating v4 vector
28              if(x < nx-1){
29                  v4.x = data[count - nx*nz + 1].x - data[count - nx*nz].x;
30                  v4.y = data[count - nx*nz + 1].y - data[count - nx*nz].y;
31                  v4.z = data[count - nx*nz + 1].z - data[count - nx*nz].z;
32              }
33
34              // Bondary check
35              if(x == 0){
36                  v2 = -v4;
37              }else if(x == nx-1){
38                  v4 = -v2;
39              }
40
41              if(z == 0){
42                  v1 = -v3;
43              }else if(z == nz-1){
44                  v3 = -v1;
45              }
46
47              // Calculating normal vector
48              v1 = glm::normalize(v1);
49              v2 = glm::normalize(v2);
50              v3 = glm::normalize(v3);
51              v4 = glm::normalize(v4);
52              n = glm::cross(v1, v2) + glm::cross(v2, v3) + glm::cross(v3, v4) + glm
                    ::cross(v4, v1);
53              n = mesh_dy*glm::normalize(n);
54              data[count].x = data[count - nx*nz].x + n.x;
55              data[count].y = data[count - nx*nz].y + n.y;
56              data[count].z = data[count - nx*nz].z + n.z;
57              data[count].w = 0.f;
58              count++;
59
60              assert(count <= num_nodes);
61          }
62      }
63  }
```

H.2 Global Displacement

One of the kernels calculating the global displacement, the other kernels are simmilar but the indexing is different.

Listing H.3: Step 1 of global displacement calculation

```
1  /**
2  * Solving displacement based on SOR, Node 1 of finite element
3  **/
4  __global__ void solve_global_displacement_step1(float3 *vertices, float3 *U,
       float3 *F, mesh_point *mesh){
5      // Varialbles
6      float3 sigma;
7      float k;
8
9      // ID
10     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
11     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
12     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
13     int i = id_y*Nx*Nz + id_z*Nx + id_x;
14
15     // Threads outside domain
16     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
17         return;
18     }
19
20     // Spring constant
21     k = spring_const(mesh, U, vertices, i);
```

```
22
23      // NO DISPLACEMENT FOR BOTTOM LAYER
24      if(id_y > 0){
25          // U[i]
26          sigma = U[i+1] + U[i+Nx] + U[i+Nx*Nz];
27          sigma = k*sigma;
28          U[i] = (1.f-relaxation)*U[i] + (relaxation/(-3.f*k))*(F[i] - sigma);
29      }
30  }
```

H.3 Propagate Fracture

Listing H.4: Step 1 of propagate fracture calculation

```
1   /**
2   * Propagating fractures in node 1 corner
3   **/
4   __global__ void propagate_fractures_step1(float3 *vertices, float3 *U, float *
        fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
        shear_stress, float3 *density){
5       // ID
6       int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
7       int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
8       int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
9       int i = id_y*Nx*Nz + id_z*Nx + id_x;
10      float dA, dE, E;
11      float a1, a2;
12      float beta = 3.1415f;
13
14      // Threads outside domain
15      if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
16          return;
17      }
18
19      // Young's modulus
20      float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
21      E = find_youngs_modulus(mesh, U, vertices, i);
22
23      // Local displacement
24      real3 u1, u2, u3, u5;
25      u1.x = U[i].x / (1.f+fracture_stress_release*fractures[i*6+0]);
26      u1.y = U[i].y / (1.f+fracture_stress_release*fractures[i*6+2]);
27      u1.z = U[i].z / (1.f+fracture_stress_release*fractures[i*6+4]);
28      u2.x = U[i+1].x / (1.f+fracture_stress_release*fractures[(i+1)*6+1]);
29      u2.y = U[i+1].y / (1.f+fracture_stress_release*fractures[(i+1)*6+2]);
30      u2.z = U[i+1].z / (1.f+fracture_stress_release*fractures[(i+1)*6+4]);
31      u3.x = U[i+Nx].x / (1.f+fracture_stress_release*fractures[(i+Nx)*6+0]);
32      u3.y = U[i+Nx].y / (1.f+fracture_stress_release*fractures[(i+Nx)*6+2]);
33      u3.z = U[i+Nx].z / (1.f+fracture_stress_release*fractures[(i+Nx)*6+5]);
34      u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*fractures[(i+Nx*Nz)*6+0]);
35      u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*fractures[(i+Nx*Nz)*6+3]);
36      u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*fractures[(i+Nx*Nz)*6+4]);
37
38      // Global vertices
39      real3 v1, v2, v3, v5;
40      v1.x = vertices[i].x;
41      v1.y = vertices[i].y;
42      v1.z = vertices[i].z;
43      v2.x = vertices[i+1].x;
44      v2.y = vertices[i+1].y;
45      v2.z = vertices[i+1].z;
46      v3.x = vertices[i+Nx].x;
47      v3.y = vertices[i+Nx].y;
48      v3.z = vertices[i+Nx].z;
49      v5.x = vertices[i+Nx*Nz].x;
50      v5.y = vertices[i+Nx*Nz].y;
51      v5.z = vertices[i+Nx*Nz].z;
52
53      // Normal strain
54      real n1, n4, n5, theta;
55      theta = dot(u2-u1, v2-v1)/(length(u2-u1) * length(v2-v1));
56      n1  = (length(u2-u1)*theta)/length(v2-v1)*E;
```

```
57      theta = dot(u3-u1, v3-v1)/(length(u3-u1) * length(v3-v1));
58      n4  = (length(u3-u1)*theta)/length(v3-v1)*E;
59      theta = dot(u5-u1, v5-v1)/(length(u5-u1) * length(v5-v1));
60      n5  = (length(u5-u1)*theta)/length(v5-v1)*E;
61
62      // Shear strain
63      real s1, s4, s5;
64      s1  = acos(dot(v2-v1, (v2+u2)-(v1+u1))/(length(v2-v1) * length((v2+u2)-(v1+u1)
            )))*E;
65      s4  = acos(dot(v3-v1, (v3+u3)-(v1+u1))/(length(v3-v1) * length((v3+u3)-(v1+u1)
            )))*E;
66      s5  = acos(dot(v5-v1, (v5+u5)-(v1+u1))/(length(v5-v1) * length((v5+u5)-(v1+u1)
            )))*E;
67
68      // XZ plane, a1 fracture = node 1 -> node 2
69      a1 = fractures[i*6+0]; // +X
70      a2 = fractures[i*6+4]; // +Z
71      dE = ((n5*n5+s4*s4+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
72      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
73      if(-(dE/dA) >= gc){
74          fractures[i*6+0] += delta_fracture;
75          a1 += delta_fracture;
76          u1.x = U[i].x / (1.f+fracture_stress_release*a1);
77      }
78
79      // XZ plane, a2 fracture = node 1 -> node 3
80      dE = ((n5*n5+s4*s4+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
81      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
82      if(-(dE/dA) >= gc){
83          fractures[i*6+4] += delta_fracture;
84          a2 += delta_fracture;
85          u1.z = U[i].z / (1.f+fracture_stress_release*a2);
86      }
87
88      // YZ plane, a1 fracture = node 1 -> node 5
89      a1 = fractures[i*6+2]; // +y
90      dE = ((n1*n1+s4*s4+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
91      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
92      if(-(dE/dA) >= gc){
93          fractures[i*6+2] += delta_fracture;
94          a1 += delta_fracture;
95          u1.y = U[i].y / (1.f+fracture_stress_release*a1);
96      }
97
98      // YZ plane, a2 fracture = node 1 -> node 3
99      dE = ((n1*n1+s4*s4+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
100     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
101     if(-(dE/dA) >= gc){
102         fractures[i*6+4] += delta_fracture;
103         a2 += delta_fracture;
104         u1.z = U[i].z / (1.f+fracture_stress_release*a2);
105     }
106
107     // YX plane, a1 fracture = node 1 -> node 5
108     a2 = fractures[i*6+0]; // +X
109     dE = ((n4*n4+s1*s1+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
110     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
111     if(-(dE/dA) >= gc){
112         fractures[i*6+2] += delta_fracture;
113         a1 += delta_fracture;
114         u1.y = U[i].y / (1.f+fracture_stress_release*a1);
115     }
116
117     // YX plane, a2 fracture = node 1 -> node 2
118     dE = ((n4*n4+s1*s1+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
```

```
            delta_fracture)));
119         dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
120         if(-(dE/dA) >= gc){
121             fractures[i*6+0] += delta_fracture;
122             a2 += delta_fracture;
123             u1.x = U[i].x / (1.f+fracture_stress_release*a2);
124         }
125
126         // Visualization
127         float temp = (-1.f)*(dE/dA) / gc;
128         if(!isnan(temp)){
129             energy[i] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
130         }
131
132         if(!isnan(abs(n1+n4+n5))){
133             normal_stress[i] = calculate_color(abs(n1+n4+n5), max_normal_stress);
134         }
135
136         if(!isnan(abs(s1+s4+s5))){
137             shear_stress[i] = calculate_color(abs(s1+s4+s5), max_shear_stress);
138         }
139
140         density[i] = calculate_color(find_density(mesh, U, vertices, i), max_density);
141     }
```

H.4 Makefile

Listing H.5: Makefile used for compilation

```
1   # This file is created by students of the HPC-Lab at the Norwegian University of
        Science and Technology (NTNU)
2   # and is part of the HPC-Lab Snow Simulator application distributed under the GPL
        license.
3   # Copyright (c) 2006-2013 High Performance Lab at the Norwegian University of
        Science and Technology (NTNU)
4   # Department of Computer and Information Science (IDI). All rights reserved.
5   # See the file README.md for more information.
6
7   CXX=g++
8   NVCC=/usr/local/cuda/bin/nvcc -m64 -arch=sm_20 --use_fast_math
9   #NVCC=/usr/local/cuda/bin/nvcc -m64 -arch=sm_35
10
11  INCLUDES=-I/usr/local/cuda/include
12  COMMONFLAGS=$(INCLUDES)
13  NVCCFLAGS=$(COMMONFLAGS)
14  CXXFLAGS=$(COMMONFLAGS) -c -Wall -O3 -std=c++0x
15
16  LIB_CUDA=-L/usr/local/cuda/lib64/ -lcudart
17  LDFLAGS=-lGL -lglfw -lGLEW -lGLU -lAntTweakBar
18
19  CSRC=soil/image_DXT.c soil/image_helper.c soil/SOIL.c soil/stb_image_aug.c
20  CPPSRC=$(shell ls *.cpp) $(shell ls image/*.cpp)
21  CUSRC=ParticleSystem.cu CudaHelpers.cu
22  OBJECTS=$(CPPSRC:.cpp=.o) $(CUSRC:.cu=.o) $(CSRC:.c=.o)
23  EXECUTABLE=snow
24
25  # Compiling ----------------------------------------------
26
27  .SUFFIXES: .c .cpp .cu .o
28  .PHONY: all run clean
29
30  all: $(EXECUTABLE)
31
32  run: $(EXECUTABLE)
33      ./snow
34
35  clean:
36      rm -f *.o snow
37
38  %c.o: %.cpp
39      $(CXX) $(CXXFLAGS) -c $< -o $@
40
```

```
%.o: %.cu
	$(NVCC) $(NVCCFLAGS) -c $< -o $@

KERNELS_WIND = WindSystemKernels.cu
SYSTEMS_WIND = WindSystem.cu
KERNELS_SNOW = SnowSystemKernels.cu
SYSTEMS_SNOW = SnowSystem.cu
KERNELS_TERR = TerrainSystemKernels.cu
SYSTEMS_TERR = TerrainSystem.cu

KERNELS = $(KERNELS_WIND) $(KERNELS_SNOW) $(KERNELS_TERR)
SYSTEMS = $(SYSTEMS_WIND) $(SYSTEMS_SNOW) $(SYSTEMS_TERR)

# Always recompile the particlesystem if subfiles are edited
ParticleSystem.o: ParticleSystem.cu $(KERNELS) $(SYSTEMS)
	$(NVCC) $(NVCCFLAGS) -c $< -o $@

# Linking -------------------------------------------------

$(EXECUTABLE): $(OBJECTS)
	$(CXX) $(OBJECTS) $(LIB_CUDA) $(LDFLAGS) -o $@
```

H.5 Accuracy Test Program

Listing H.6: Floating point accuracy test program

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef double number;

int main(int argc, char **args){
    // V1
    number x1 = 1.f;
    number y1 = 0.f;
    number z1 = 0.f;

    // V2
    number x2 = 2.f;
    number y2 = 0.f;
    number z2 = 0.f;

    // U1
    number u1x = 0.f;
    number u1y = 0.00000001f;
    number u1z = 0.f;

    // U2
    number u2x = 0.f;
    number u2y = 0.f;
    number u2z = 0.f;

    // Orginal vec
    number vec1x = x2 - x1;
    number vec1y = y2 - y1;
    number vec1z = z2 - z1;

    // Vec with displacement
    number vec2x = (x2+u2x) - (x1+u1x);
    number vec2y = (y2+u2y) - (y1+u1y);
    number vec2z = (z2+u2z) - (z1+u1z);

    // Total displacement
    number vec3x = u1x-u2x;
    number vec3y = u1y-u2y;
    number vec3z = u1z-u2z;

    // Shear
    number E = 12.f * 1000000.f;
    number nom = vec1x*vec2x + vec1y*vec2y + vec1z*vec2z;
```

```
46      number dom = sqrt(vec1x*vec1x + vec1y*vec1y + vec1z*vec1z) * sqrt(vec2x*vec2x
            + vec2y*vec2y + vec2z*vec2z);
47      number strain = acos(nom/dom);
48      printf("shear␣strain:␣%f␣/␣%f␣=␣%f\n", nom, dom, strain);
49      printf("shear␣stress:␣%f\n\n", strain*E);
50
51      // Normal
52      number theta = ((vec3x*vec1x) + (vec3y*vec1y) + (vec3z*vec1z))/(sqrt(vec3x*
            vec3x + vec3y*vec3y + vec3z*vec3z) * sqrt((vec1x*vec1x) + (vec1y*vec1y) +
            (vec1z*vec1z)));
53      nom = sqrt(vec3x*vec3x + vec3y*vec3y + vec3z*vec3z)*theta;
54      dom = sqrt(vec1x*vec1x + vec1y*vec1y + vec1z*vec1z);
55      strain = nom/dom;
56      printf("Normal␣strain:␣%f␣/␣%f␣=␣%f\n", nom, dom, strain);
57      printf("Normal␣stress:␣%f\n", strain*E);
58
59      return EXIT_SUCCESS;
60  }
```

H.6 CPU Version

Compiled with *g++ -o main -O3 main.c -lm -lGL -fopenmp*

Listing H.7: Floating point accuracy test program

```
1   // Time
2   timeval t0, t1;
3   gettimeofday(&t0, NULL);
4
5   int i,j,k;
6   count = 0;
7   while(count < time_steps){
8     // Increase timestep
9     count++;
10
11    // Solve displacement
12    #pragma omp parallel for
13    for(i=0; i<Nx; i++){
14      for(j=0; j<Ny; j++){
15        for(k=0; k<Nz; k++){
16          // Solving displacement
17          solve_global_displacement_step1(i, j, k, vertices, U, F);
18          solve_global_displacement_step2(i, j, k, vertices, U, F);
19          solve_global_displacement_step3(i, j, k, vertices, U, F);
20          solve_global_displacement_step4(i, j, k, vertices, U, F);
21          solve_global_displacement_step5(i, j, k, vertices, U, F);
22          solve_global_displacement_step6(i, j, k, vertices, U, F);
23          solve_global_displacement_step7(i, j, k, vertices, U, F);
24          solve_global_displacement_step8(i, j, k, vertices, U, F);
25
26          propagate_fractures_step1(i, j, k, vertices, U, fractures);
27          propagate_fractures_step2(i, j, k, vertices, U, fractures);
28          propagate_fractures_step3(i, j, k, vertices, U, fractures);
29          propagate_fractures_step4(i, j, k, vertices, U, fractures);
30          propagate_fractures_step5(i, j, k, vertices, U, fractures);
31          propagate_fractures_step6(i, j, k, vertices, U, fractures);
32          propagate_fractures_step7(i, j, k, vertices, U, fractures);
33          propagate_fractures_step8(i, j, k, vertices, U, fractures);
34        }
35      }
36    }
37  }
38
39  gettimeofday(&t1, NULL);
40  long int micro = ((t1.tv_sec t0.tv_sec)*1000000L+t1.tv_usec)-t0.tv_usec;
41  printf("time␣%f␣sec\n", micro/1000000.f);
42  return EXIT_SUCCESS;
```

H.7 Complete Code

Listing H.8: Complete code

```
1  // This file is created by students of the HPC-Lab at the Norwegian University of
       Science and Technology (NTNU)
2  // and is part of the HPC-Lab Snow Simulator application distributed under the GPL
       license.
3  // Copyright (c) 2006-2013 High Performance Lab at the Norwegian University of
       Science and Technology (NTNU)
4  // Department of Computer and Information Science (IDI). All rights reserved.
5  // See the file README.md for more information.
6
7  // Used to switch between float and double for stress calculation
8  typedef double real;
9  typedef double3 real3;
10
11 __constant__ float cuda_max_snow_height;
12 __constant__ float cuda_mesh_dx;
13 __constant__ float cuda_mesh_dy;
14 __constant__ float cuda_mesh_dz;
15 __constant__ float delta_fracture;
16
17 // Nodes dimension
18 __constant__ int Nx;
19 __constant__ int Ny;
20 __constant__ int Nz;
21
22 // Number of elements
23 __constant__ int Ex;
24 __constant__ int Ey;
25 __constant__ int Ez;
26
27 // SOR
28 __constant__ float relaxation;
29
30 const float fracture_stress_release = 1000.f;
31 const float max_normal_stress = 10000.f;
32 const float max_shear_stress = 10000.f;
33 const float max_density = 1000.f;
34
35 /**
36 * Inline functions
37 * found at http://bullet.googlecode.com/svn/trunk/Extras/CUDA/cutil_math.h
38 * Contains:
39 *        - cross:        vector cross product
40 *        - length:        length of a vector
41 *        - dot:            dot product between two vectors
42 *        - operator+:    vector3 addition operation
43 *        - operator/:    vector3 division operation
44 *        - operator-:    vector3 subtraction operation
45 *        - operator*:    vector3 multiplication operation
46 **/
47 inline __host__ __device__ float3 cross(float3 a, float3 b)
48 {
49     return make_float3(a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x);
50 }
51 inline __host__ __device__ float dot(float3 a, float3 b)
52 {
53     return a.x * b.x + a.y * b.y + a.z * b.z;
54 }
55 inline __host__ __device__ float length(float3 v)
56 {
57     return sqrtf(dot(v, v));
58 }
59 inline __host__ __device__ double dot(double3 a, double3 b)
60 {
61     return a.x * b.x + a.y * b.y + a.z * b.z;
62 }
63 inline __host__ __device__ double length(double3 v)
64 {
65     return sqrt(dot(v, v));
66 }
67 inline __host__ __device__ float3 operator+(float3 a, float3 b)
68 {
69     return make_float3(a.x + b.x, a.y + b.y, a.z + b.z);
70 }
71 inline __host__ __device__ float3 operator/(float3 a, float3 b)
72 {
```

```
73      return make_float3(a.x / b.x, a.y / b.y, a.z / b.z);
74  }
75  inline __host__ __device__ float3 operator-(float3 a, float3 b)
76  {
77      return make_float3(a.x - b.x, a.y - b.y, a.z - b.z);
78  }
79  inline __host__ __device__ float3 operator*(float3 a, float3 b)
80  {
81      return make_float3(a.x * b.x, a.y * b.y, a.z * b.z);
82  }
83  inline __host__ __device__ float3 operator*(float3 a, float b)
84  {
85      return make_float3(a.x * b, a.y * b, a.z * b);
86  }
87  inline __host__ __device__ float3 operator*(float a, float3 b)
88  {
89      return make_float3(a * b.x, a * b.y, a * b.z);
90  }
91  inline __host__ __device__ double3 operator+(double3 a, double3 b)
92  {
93      return make_double3(a.x + b.x, a.y + b.y, a.z + b.z);
94  }
95  inline __host__ __device__ double3 operator-(double3 a, double3 b)
96  {
97      return make_double3(a.x - b.x, a.y - b.y, a.z - b.z);
98  }
99
100 /**
101 * Calculates the RGB color from a HSV color
102 * H = 240 - values
103 * S = 1
104 * V = 1
105 * http://www.cs.rit.edu/~ncs/color/t_convert.html
106 **/
107 __device__ float3 calculate_color(float value, float max){
108     int i;
109     float f, p, q, t;
110     float s = 1.f;
111     float v = 1.f;
112     float h = 240.f - 240.f*(value/max);
113
114     if(value > max){value = max;}
115
116     h /= 60;            // sector 0 to 5
117     i = floor(h);
118     f = h - i;            // factorial part of h
119     p = v * (1 - s);
120     q = v * (1 - s*f);
121     t = v * (1 - s*(1 - f));
122     switch( i ) {
123         case 0:
124             return make_float3(v, t, p);
125         case 1:
126             return make_float3(q, v, p);
127         case 2:
128             return make_float3(p, v, t);
129         case 3:
130             return make_float3(p, q, v);
131         case 4:
132             return make_float3(t, p, v);
133         default:        // case 5:
134             return make_float3(v, p, q);
135     }
136 }
137
138 /**
139 * Function for finding the density of a finite element
140 **/
141 __device__ float find_density(mesh_point *mesh_points, float3 *U, float3 *vertices
        , int i){
142     // Calculating volume
143     float3 v1 = vertices[i] + U[i];
144     float3 v2 = vertices[i+1] + U[i+1];
145     float3 v3 = vertices[i+Nx] + U[i+Nx];
146     float3 v4 = vertices[i+Nx+1] + U[i+Nx+1];
147     float3 v5 = vertices[i+Nx*Nz] + U[i+Nx*Nz];
```

```
148     float3 v6 = vertices[i+Nx*Nz+1] + U[i+Nx*Nz+1];
149     float3 v7 = vertices[i+Nx*Nz+Nx] + U[i+Nx*Nz+Nx];
150     float3 v8 = vertices[i+Nx*Nz+Nx+1] + U[i+Nx*Nz+Nx+1];
151     float t1 = (dot(v1-v5, cross(v2-v5, v3-v5)))/6.f;
152     float t2 = (dot(v2-v8, cross(v3-v8, v4-v8)))/6.f;
153     float t3 = (dot(v2-v8, cross(v5-v8, v6-v8)))/6.f;
154     float t4 = (dot(v3-v8, cross(v5-v8, v7-v8)))/6.f;
155     float t5 = (dot(v2-v8, cross(v3-v8, v5-v8)))/6.f;
156     float volume = t1 + t2 + t3 + t4 + t5;
157
158     // Element temperature and humidity
159     //mesh_point point = mesh_points[id];
160     //unsigned char temp_temperature = point.type & 0xf8;
161     //unsigned char temp_humidity = point.type & 0x07;
162     //int temperature = -(temp_temperature >> 3); // Is now in range [-1, -31]
163     //float humidity = ((temp_humidity * 1.f) / 7.f) * 0.3f; // Is now in range
            [0.042, 0.3]
164     float mass = 20.f; // Using constant mass i.e. homogeneous snow
165
166     // Density = mass/volume
167     return mass/volume;
168 }
169
170 /**
171 * Function for finding Young's modulus for a given mesh element (Will need
        research for being 100% correct)
172 **/
173 __device__ float find_youngs_modulus(mesh_point *mesh_points, float3 *U, float3 *
        vertices, int id){
174     return 60.f * 1000000.f; // 60 MPa
175     //return 1.89 * powf(find_density(mesh_points, U, vertices, id), 2.94f);
176 }
177
178 /**
179 * Function for finding the critical energy release rate for a given finite element
180 **/
181 __device__ float find_critical_energy_release_rate(mesh_point *mesh_points, float3
         *U, float3 *vertices, int id){
182     float k = 0.00042f * powf(find_density(mesh_points, U, vertices, id), 2.76f);
183     return (k*k)/find_youngs_modulus(mesh_points, U, vertices, id);
184     //return 0.044;
185 }
186
187 /**
188 * Function for finding spring constant for a given finite element (Will need
        research for being 100% correct)
189 **/
190 __device__ float spring_const(mesh_point *mesh_points, float3 *U, float3 *vertices
        , int id){
191     return 10000.f*find_density(mesh_points, U, vertices, id);
192 }
193
194 /**
195 * Fills the snow grid data with current air temperature and humidity, IF covered
        by snow
196 *
197 * Params:
198 *     - grid:                Height data of the snow and terrain
199 *     - mesh_points:        The 3D mesh points
200 *     - air_temperature:    Current air temperature
201 *     - air_humidity:        Current air humidity
202 *
203 * air_humidity range = [0, 0.3]
204 * air_temperature range = [-1, -31]
205 */
206 __global__ void fill_mesh(float4 *grid, mesh_point *mesh_points, float
        air_temperature, float air_humidity, int *max_filled_snow_layer) {
207     //Init
208     int gtx = (blockIdx.x * blockDim.x) + threadIdx.x;
209     int gty = (blockIdx.y * blockDim.y) + threadIdx.y;
210     int bx = gtx * cuda_mesh_dx;
211     int by = gty * cuda_mesh_dz;
212
213     float snow_height = grid[(by * terrain_dim) + bx].w;
214     int upper_mesh_id = (int)((snow_height / cuda_mesh_dy));
215     upper_mesh_id = upper_mesh_id - 1;
```

```
216
217         //Snow has reached at least 1 grid point
218         if(upper_mesh_id >= 0 && snow_height >= cuda_mesh_dy && upper_mesh_id <
                cuda_max_snow_height / cuda_mesh_dy){
219             //Domain size
220             int sizeX = (int)(terrain_dim / cuda_mesh_dx);
221             int sizeY = (int)(terrain_dim / cuda_mesh_dz);
222             int snow_mesh_id = (upper_mesh_id * sizeY * sizeX) + (gty * sizeX) + gtx;
223
224             // Setting max_snow_layer
225             if(upper_mesh_id > *max_filled_snow_layer){
226                 *max_filled_snow_layer = upper_mesh_id;
227             }
228
229             if(mesh_points[snow_mesh_id].type == 0){
230                 //Setting mesh data
231                 unsigned char temperature = (unsigned char)((-1.f) * air_temperature);
232                 unsigned char humidity = (unsigned char)((air_humidity / 0.3f) * 7.f);
233
234                 temperature = temperature << 3;
235                 mesh_points[snow_mesh_id].type = temperature | humidity;
236             }
237         }
238     }
239
240     /**
241     * Solving displacement based on SOR, Node 1 of finite element
242     **/
243     __global__ void solve_global_displacement_step1(float3 *vertices, float3 *U,
            float3 *F, mesh_point *mesh){
244         // Varialbles
245         float3 sigma;
246         float k;
247
248         // ID
249         int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
250         int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
251         int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
252         int i = id_y*Nx*Nz + id_z*Nx + id_x;
253
254         // Threads outside domain
255         if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
256             return;
257         }
258
259         // Spring constant
260         k = spring_const(mesh, U, vertices, i);
261
262         // NO DISPLACEMENT FOR BOTTOM LAYER
263         if(id_y > 0){
264             // U[i]
265             sigma = U[i+1] + U[i+Nx] + U[i+Nx*Nz];
266             sigma = k*sigma;
267             U[i] = (1.f-relaxation)*U[i] + (relaxation/(-3.f*k))*(F[i] - sigma);
268         }
269     }
270
271     /**
272     * Solving displacement based on SOR, Node 2 of finite element
273     **/
274     __global__ void solve_global_displacement_step2(float3 *vertices, float3 *U,
            float3 *F, mesh_point *mesh){
275         // Varialbles
276         float3 sigma;
277         float k;
278
279         // ID
280         id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
281         int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
282         int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
283         int i = id_y*Nx*Nz + id_z*Nx + id_x;
284
285         // Threads outside domain
286         if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
287             return;
288         }
```

```
289
290     // Spring constant
291     k = spring_const(mesh, U, vertices, i);
292
293     if(id_y > 0){
294         // U[i+1]
295         sigma = U[i] + U[i+Nx+1] + U[i+Nx*Nz+1];
296         sigma = k*sigma;
297         U[i+1] = (1.f-relaxation)*U[i+1] + (relaxation/(-3.f*k))*(F[i+1] - sigma);
298     }
299 }
300
301 /**
302 * Solving displacement based on SOR, Node 3 of finite element
303 **/
304 __global__ void solve_global_displacement_step3(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
305     // Varialbles
306     float3 sigma;
307     float k;
308
309     // ID
310     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
311     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
312     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
313     int i = id_y*Nx*Nz + id_z*Nx + id_x;
314
315     // Threads outside domain
316     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
317         return;
318     }
319
320     // Spring constant
321     k = spring_const(mesh, U, vertices, i);
322
323     if(id_y > 0){
324         // U[i+Nx]
325         sigma = U[i] + U[i+Nx+1] + U[i+Nx*Nz+Nx];
326         sigma = k*sigma;
327         U[i+Nx] = (1.f-relaxation)*U[i+Nx] + (relaxation/(-3.f*k))*(F[i+Nx] -
                sigma);
328     }
329 }
330
331 /**
332 * Solving displacement based on SOR, Node 4 of finite element
333 **/
334 __global__ void solve_global_displacement_step4(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
335     // Varialbles
336     float3 sigma;
337     float k;
338
339     // ID
340     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
341     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
342     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
343     int i = id_y*Nx*Nz + id_z*Nx + id_x;
344
345     // Threads outside domain
346     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
347         return;
348     }
349
350     // Spring constant
351     k = spring_const(mesh, U, vertices, i);
352
353     if(id_y > 0){
354         // U[i+Nx+1]
355         sigma = U[i+1] + U[i+Nx] + U[i+Nx*Nz+Nx+1];
356         sigma = k*sigma;
357         U[i+Nx+1] = (1.f-relaxation)*U[i+Nx+1] + (relaxation/(-3.f*k))*(F[i+Nx+1]
                - sigma);
358     }
359 }
360
```

```
361 /**
362 * Solving displacement based on SOR, Node 5 of finite element
363 **/
364 __global__ void solve_global_displacement_step5(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
365     // Varialbles
366     float3 sigma;
367     float k;
368
369     // ID
370     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
371     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
372     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
373     int i = id_y*Nx*Nz + id_z*Nx + id_x;
374
375     // Threads outside domain
376     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
377         return;
378     }
379
380     // Spring constant
381     k = spring_const(mesh, U, vertices, i);
382
383     // U[i+Nx*Nz]
384     sigma = U[i] + U[i+Nx*Nz+1] + U[i+Nx*Nz+Nx];
385     sigma = k*sigma;
386     U[i+Nx*Nz] = (1.f-relaxation)*U[i+Nx*Nz] + (relaxation/(-3.f*k))*(F[i+Nx*Nz] -
            sigma);
387 }
388
389 /**
390 * Solving displacement based on SOR, Node 6 of finite element
391 **/
392 __global__ void solve_global_displacement_step6(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
393     // Varialbles
394     float3 sigma;
395     float k;
396
397     // ID
398     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
399     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
400     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
401     int i = id_y*Nx*Nz + id_z*Nx + id_x;
402
403     // Threads outside domain
404     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
405         return;
406     }
407
408     // Spring constant
409     k = spring_const(mesh, U, vertices, i);
410
411     // U[i+Nx*Nz+1]
412     sigma = U[i+1] + U[i+Nx*Nz] + U[i+Nx*Nz+Nx+1];
413     sigma = k*sigma;
414     U[i+Nx*Nz+1] = (1.f-relaxation)*U[i+Nx*Nz+1] + (relaxation/(-3.f*k))*(F[i+Nx*
            Nz+1] - sigma);
415 }
416
417 /**
418 * Solving displacement based on SOR, Node 7 of finite element
419 **/
420 __global__ void solve_global_displacement_step7(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
421     // Varialbles
422     float3 sigma;
423     float k;
424
425     // ID
426     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
427     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
428     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
429     int i = id_y*Nx*Nz + id_z*Nx + id_x;
430
431     // Threads outside domain
```

```
432     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
433         return;
434     }
435 
436     // Spring constant
437     k = spring_const(mesh, U, vertices, i);
438 
439     // U[i+Nx*Nz+Nx]
440     sigma = U[i+Nx] + U[i+Nx*Nz] + U[i+Nx*Nz+Nx+1];
441     sigma = k*sigma;
442     U[i+Nx*Nz+Nx] = (1.f-relaxation)*U[i+Nx*Nz+Nx] + (relaxation/(-3.f*k))*(F[i+Nx
            *Nz+Nx] - sigma);
443 }
444 
445 /**
446 * Solving displacement based on SOR, Node 8 of finite element
447 **/
448 __global__ void solve_global_displacement_step8(float3 *vertices, float3 *U,
        float3 *F, mesh_point *mesh){
449     // Varialbles
450     float3 sigma;
451     float k;
452 
453     // ID
454     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
455     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
456     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
457     int i = id_y*Nx*Nz + id_z*Nx + id_x;
458 
459     // Threads outside domain
460     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
461         return;
462     }
463 
464     // Spring constant
465     k = spring_const(mesh, U, vertices, i);
466 
467     // U[i+Nx*Nz+Nx+1]
468     sigma = U[i+Nx+1] + U[i+Nx*Nz+1] + U[i+Nx*Nz+Nx];
469     sigma = k*sigma;
470     U[i+Nx*Nz+Nx+1] = (1.f-relaxation)*U[i+Nx*Nz+Nx+1] + (relaxation/(-3.f*k))*(F[
            i+Nx*Nz+Nx+1] - sigma);
471 }
472 
473 /**
474 * Propagating fractures in node 1 corner
475 **/
476 __global__ void propagate_fractures_step1(float3 *vertices, float3 *U, float *
        fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
        shear_stress, float3 *density){
477     // ID
478     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
479     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
480     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
481     int i = id_y*Nx*Nz + id_z*Nx + id_x;
482     float dA, dE, E;
483     float a1, a2;
484     float beta = 3.1415f;
485 
486     // Threads outside domain
487     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
488         return;
489     }
490 
491     // Young's modulus
492     float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
493     E = find_youngs_modulus(mesh, U, vertices, i);
494 
495     // Local displacement
496     real3 u1, u2, u3, u5;
497     u1.x = U[i].x / (1.f+fracture_stress_release*(fractures[i*6+2]+fractures[i
            *6+4]));
498     u1.y = U[i].y / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
            *6+4]));
499     u1.z = U[i].z / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
            *6+2]));
```

```
500     u2.x = U[i+1].x / (1.f+fracture_stress_release*(fractures[(i+1)*6+2]+fractures
            [(i+1)*6+4]));
501     u2.y = U[i+1].y / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+4]));
502     u2.z = U[i+1].z / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+2]));
503     u3.x = U[i+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+2]+
            fractures[(i+Nx)*6+5]));
504     u3.y = U[i+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
            fractures[(i+Nx)*6+5]));
505     u3.z = U[i+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
            fractures[(i+Nx)*6+2]));
506     u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+3]+
            fractures[(i+Nx*Nz)*6+4]));
507     u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
            fractures[(i+Nx*Nz)*6+4]));
508     u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
            fractures[(i+Nx*Nz)*6+3]));
509
510     // Global vertices
511     real3 v1, v2, v3, v5;
512     v1.x = vertices[i].x;
513     v1.y = vertices[i].y;
514     v1.z = vertices[i].z;
515     v2.x = vertices[i+1].x;
516     v2.y = vertices[i+1].y;
517     v2.z = vertices[i+1].z;
518     v3.x = vertices[i+Nx].x;
519     v3.y = vertices[i+Nx].y;
520     v3.z = vertices[i+Nx].z;
521     v5.x = vertices[i+Nx*Nz].x;
522     v5.y = vertices[i+Nx*Nz].y;
523     v5.z = vertices[i+Nx*Nz].z;
524
525     // Normal strain
526     real n1, n4, n5, theta;
527     theta = dot(u2-u1, v2-v1)/(length(u2-u1) * length(v2-v1));
528     n1  = (length(u2-u1)*theta)/length(v2-v1)*E;
529     theta = dot(u3-u1, v3-v1)/(length(u3-u1) * length(v3-v1));
530     n4  = (length(u3-u1)*theta)/length(v3-v1)*E;
531     theta = dot(u5-u1, v5-v1)/(length(u5-u1) * length(v5-v1));
532     n5  = (length(u5-u1)*theta)/length(v5-v1)*E;
533
534     // Shear strain
535     real s1, s4, s5;
536     s1  = acos(dot(v2-v1, (v2+u2)-(v1+u1))/(length(v2-v1) * length((v2+u2)-(v1+u1)
            )))*E;
537     s4  = acos(dot(v3-v1, (v3+u3)-(v1+u1))/(length(v3-v1) * length((v3+u3)-(v1+u1)
            )))*E;
538     s5  = acos(dot(v5-v1, (v5+u5)-(v1+u1))/(length(v5-v1) * length((v5+u5)-(v1+u1)
            )))*E;
539
540     // XZ plane, a1 fracture = node 1 -> node 2
541     a1 = fractures[i*6+0]; // +X
542     a2 = fractures[i*6+4]; // +Z
543     dE = ((n5*n5+s4*s4+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
544     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
545     if(-(dE/dA) >= gc){
546         fractures[i*6+0] += delta_fracture;
547         a1 += delta_fracture;
548         u1.x = U[i].x / (1.f+fracture_stress_release*a1);
549     }
550
551     // XZ plane, a2 fracture = node 1 -> node 3
552     dE = ((n5*n5+s4*s4+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
553     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
554     if(-(dE/dA) >= gc){
555         fractures[i*6+4] += delta_fracture;
556         a2 += delta_fracture;
557         u1.z = U[i].z / (1.f+fracture_stress_release*a2);
558     }
559
```

```
560     // YZ plane, a1 fracture = node 1 -> node 5
561     a1 = fractures[i*6+2]; // +y
562     dE = ((n1*n1+s4*s4+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
563     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
564     if(-(dE/dA) >= gc){
565         fractures[i*6+2] += delta_fracture;
566         a1 += delta_fracture;
567         u1.y = U[i].y / (1.f+fracture_stress_release*a1);
568     }
569
570     // YZ plane, a2 fracture = node 1 -> node 3
571     dE = ((n1*n1+s4*s4+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
572     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
573     if(-(dE/dA) >= gc){
574         fractures[i*6+4] += delta_fracture;
575         a2 += delta_fracture;
576         u1.z = U[i].z / (1.f+fracture_stress_release*a2);
577     }
578
579     // YX plane, a1 fracture = node 1 -> node 5
580     a2 = fractures[i*6+0]; // +X
581     dE = ((n4*n4+s1*s1+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
582     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
583     if(-(dE/dA) >= gc){
584         fractures[i*6+2] += delta_fracture;
585         a1 += delta_fracture;
586         u1.y = U[i].y / (1.f+fracture_stress_release*a1);
587     }
588
589     // YX plane, a2 fracture = node 1 -> node 2
590     dE = ((n4*n4+s1*s1+s5*s5)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
591     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
592     if(-(dE/dA) >= gc){
593         fractures[i*6+0] += delta_fracture;
594         a2 += delta_fracture;
595         u1.x = U[i].x / (1.f+fracture_stress_release*a2);
596     }
597
598     // Visualization
599     float temp = (-1.f)*(dE/dA) / gc;
600     if(!isnan(temp)){
601         energy[i] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
602     }
603
604     if(!isnan(abs(n1+n4+n5))){
605         normal_stress[i] = calculate_color(abs(n1+n4+n5), max_normal_stress);
606     }
607
608     if(!isnan(abs(s1+s4+s5))){
609         shear_stress[i] = calculate_color(abs(s1+s4+s5), max_shear_stress);
610     }
611
612     density[i] = calculate_color(find_density(mesh, U, vertices, i), max_density);
613 }
614
615 /**
616 * Propagating fractures in node 2 corner
617 **/
618 __global__ void propagate_fractures_step2(float3 *vertices, float3 *U, float *
        fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
        shear_stress, float3 *density){
619     // ID
620     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
621     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
622     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
623     int i = id_y*Nx*Nz + id_z*Nx + id_x;
624     float dA, dE, E;
625     float a1, a2;
```

```
626     float beta = 3.1415f;
627
628     // Threads outside domain
629     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
630         return;
631     }
632
633     // Young's modulus
634     float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
635     E = find_youngs_modulus(mesh, U, vertices, i);
636
637     // Local displacement
638     real3 u1, u2, u4, u6;
639     u1.x = U[i].x / (1.f+fracture_stress_release*(fractures[i*6+2]+fractures[i
            *6+4]));
640     u1.y = U[i].y / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
            *6+4]));
641     u1.z = U[i].z / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
            *6+2]));
642     u2.x = U[i+1].x / (1.f+fracture_stress_release*(fractures[(i+1)*6+2]+fractures
            [(i+1)*6+4]));
643     u2.y = U[i+1].y / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+4]));
644     u2.z = U[i+1].z / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+2]));
645     u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+2]+
            fractures[(i+Nx+1)*6+5]));
646     u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
            fractures[(i+Nx+1)*6+5]));
647     u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
            fractures[(i+Nx+1)*6+2]));
648     u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
            *6+3]+fractures[(i+Nx*Nz+1)*6+4]));
649     u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
            *6+1]+fractures[(i+Nx*Nz+1)*6+4]));
650     u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
            *6+1]+fractures[(i+Nx*Nz+1)*6+3]));
651
652     // Global vertices
653     real3 v1, v2, v4, v6;
654     v1.x = vertices[i].x;
655     v1.y = vertices[i].y;
656     v1.z = vertices[i].z;
657     v2.x = vertices[i+1].x;
658     v2.y = vertices[i+1].y;
659     v2.z = vertices[i+1].z;
660     v4.x = vertices[i+Nx+1].x;
661     v4.y = vertices[i+Nx+1].y;
662     v4.z = vertices[i+Nx+1].z;
663     v6.x = vertices[i+Nx*Nx+1].x;
664     v6.y = vertices[i+Nx*Nx+1].y;
665     v6.z = vertices[i+Nx*Nx+1].z;
666
667     // Normal strain
668     real n1, n2, n6, theta;
669     theta = dot(u2-u1, v2-v1)/(length(u2-u1) * length(v2-v1));
670     n1  = (length(u2-u1)*theta)/length(v2-v1)*E;
671     theta = dot(u4-u2, v4-v2)/(length(u4-u2) * length(v4-v2));
672     n2  = (length(u4-u2)*theta)/length(v4-v2)*E;
673     theta = dot(u6-u2, v6-v2)/(length(u6-u2) * length(v6-v2));
674     n6  = (length(u6-u2)*theta)/length(v6-v2)*E;
675
676     // Shear strain
677     real s1, s2, s6;
678     s1  = acos(dot(v2-v1, (v2+u2)-(v1+u1))/(length(v2-v1) * length((v2+u2)-(v1+u1)
            )))*E;
679     s2  = acos(dot(v4-v2, (v4+u4)-(v2+u2))/(length(v4-v2) * length((v4+u4)-(v2+u2)
            )))*E;
680     s6  = acos(dot(v6-v2, (v6+u6)-(v2+u2))/(length(v6-v2) * length((v6+u6)-(v2+u2)
            )))*E;
681
682     // XZ plane, a1 fracture = node 2 -> node 1
683     a1 = fractures[(i+1)*6+1]; // -X
684     a2 = fractures[(i+1)*6+4]; // +Z
685     dE = ((n6*n6+s1*s1+s2*s2)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
```

```
        *a2));
686     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
687     if(-(dE/dA) >= gc){
688         fractures[(i+1)*6+1] += delta_fracture;
689         a1 += delta_fracture;
690         u2.x = U[i+1].x / (1.f+fracture_stress_release*a1);
691     }
692
693     // XZ plane, a2 fracture = node 2 -> node 4
694     dE = ((n6*n6+s1*s1+s2*s2)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
        sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
        delta_fracture)));
695     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
696     if(-(dE/dA) >= gc){
697         fractures[(i+1)*6+4] += delta_fracture;
698         a2 += delta_fracture;
699         u2.z = U[i+1].z / (1.f+fracture_stress_release*a2);
700     }
701
702     // YZ plane, a1 fracture = node 2 -> node 6
703     a1 = fractures[(i+1)*6+2]; // +Y
704     dE = ((n1*n1+s6*s6+s2*s2)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
        sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
        *a2));
705     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
706     if(-(dE/dA) >= gc){
707         fractures[(i+1)*6+2] += delta_fracture;
708         a1 += delta_fracture;
709         u2.y = U[i+1].y / (1.f+fracture_stress_release*a1);
710     }
711
712     // YZ plane, a2 fracture = node 2 -> node 4
713     dE = ((n1*n1+s6*s6+s2*s2)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
        sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
        *a2));
714     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
715     if(-(dE/dA) >= gc){
716         fractures[(i+1)*6+4] += delta_fracture;
717         a2 += delta_fracture;
718         u2.z = U[i+1].z / (1.f+fracture_stress_release*a2);
719     }
720
721     // YX plane, a1 fracture = node 2 -> node 6
722     a2 = fractures[(i+1)*6+1]; // -X
723     dE = ((n2*n2+s6*s6+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
        sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
        *a2));
724     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
725     if(-(dE/dA) >= gc){
726         fractures[(i+1)*6+2] += delta_fracture;
727         a2 += delta_fracture;
728         u2.y = U[i+1].y / (1.f+fracture_stress_release*a2);
729     }
730
731     // YX plane a2 fracture = node 2 -> node 1
732     dE = ((n2*n2+s6*s6+s1*s1)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
        sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
        *a2));
733     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
734     if(-(dE/dA) >= gc){
735         fractures[(i+1)*6+1] += delta_fracture;
736         a2 += delta_fracture;
737         u2.x = U[i+1].x / (1.f+fracture_stress_release*a2);
738     }
739
740     // Visualization
741     float temp = (-1.f)*(dE/dA) / gc;
742     if(!isnan(temp)){
743         energy[i+1] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
744     }
745
746     if(!isnan(abs(n1+n2+n6))){
747         normal_stress[i+1] = calculate_color(abs(n1+n2+n6), max_normal_stress);
748     }
749
750     if(!isnan(abs(s1+s2+s6))){
```

```
751             shear_stress[i+1] = calculate_color(abs(s1+s2+s6), max_shear_stress);
752         }
753 
754         density[i+1] = calculate_color(find_density(mesh, U, vertices, i), max_density
                );
755     }
756 
757     /**
758     * Propagating fractures in node 3 corner
759     **/
760     __global__ void propagate_fractures_step3(float3 *vertices, float3 *U, float *
            fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
            shear_stress, float3 *density){
761         // ID
762         int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
763         int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
764         int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
765         int i = id_y*Nx*Nz + id_z*Nx + id_x;
766         float dA, dE, E;
767         float a1, a2;
768         float beta = 3.1415f;
769 
770         // Threads outside domain
771         if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
772             return;
773         }
774 
775         // Young's modulus
776         float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
777         E = find_youngs_modulus(mesh, U, vertices, i);
778 
779         // Local displacement
780         real3 u1, u3, u4, u7;
781         u1.x = U[i].x / (1.f+fracture_stress_release*(fractures[i*6+2]+fractures[i
                *6+4]));
782         u1.y = U[i].y / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
                *6+4]));
783         u1.z = U[i].z / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
                *6+2]));
784         u3.x = U[i+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+2]+
                fractures[(i+Nx)*6+5]));
785         u3.y = U[i+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
                fractures[(i+Nx)*6+5]));
786         u3.z = U[i+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
                fractures[(i+Nx)*6+2]));
787         u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+2]+
                fractures[(i+Nx+1)*6+5]));
788         u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
                fractures[(i+Nx+1)*6+5]));
789         u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
                fractures[(i+Nx+1)*6+2]));
790         u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+3]+fractures[(i+Nx*Nz+Nx)*6+5]));
791         u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+5]));
792         u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+3]));
793 
794         // Global vertices
795         real3 v1, v3, v4, v7;
796         v1.x = vertices[i].x;
797         v1.y = vertices[i].y;
798         v1.z = vertices[i].z;
799         v3.x = vertices[i+Nx].x;
800         v3.y = vertices[i+Nx].y;
801         v3.z = vertices[i+Nx].z;
802         v4.x = vertices[i+Nx+1].x;
803         v4.y = vertices[i+Nx+1].y;
804         v4.z = vertices[i+Nx+1].z;
805         v7.x = vertices[i+Nx*Nz+Nx].x;
806         v7.y = vertices[i+Nx*Nz+Nx].y;
807         v7.z = vertices[i+Nx*Nz+Nx].z;
808 
809         // Normal strain
810         real n3, n4, n8, theta;
811         theta = dot(u4-u3, v4-v3)/(length(u4-u3) * length(v4-v3));
```

```
812     n3  = (length(u4-u3)*theta)/length(v4-v3)*E;
813     theta = dot(u3-u1, v3-v1)/(length(u3-u1) * length(v3-v1));
814     n4  = (length(u3-u1)*theta)/length(v3-v1)*E;
815     theta = dot(u7-u3, v7-v3)/(length(u7-u3) * length(v7-v3));
816     n8  = (length(u7-u3)*theta)/length(v7-v3)*E;
817
818     // Shear strain
819     real s3, s4, s8;
820     s3  = acos(dot(v4-v3, (v4+u4)-(v3+u3))/(length(v4-v3) * length((v4+u4)-(v3+u3)
            )))*E;
821     s4  = acos(dot(v3-v1, (v3+u3)-(v1+u1))/(length(v3-v1) * length((v3+u3)-(v1+u1)
            )))*E;
822     s8  = acos(dot(v7-v3, (v7+u7)-(v3+u3))/(length(v7-v3) * length((v7+u7)-(v3+u3)
            )))*E;
823
824     // XZ plane, a1 fracture = node 3 -> node 1
825     a1 = fractures[(i+Nx)*6+5]; // -Z
826     a2 = fractures[(i+Nx)*6+0]; // +X
827     dE = ((n8*n8+s3*s3+s4*s4)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
828     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
829     if(-(dE/dA) >= gc){
830         fractures[(i+Nx)*6+5] += delta_fracture;
831         a1 += delta_fracture;
832         u3.z = U[i+Nx].z / (1.f+fracture_stress_release*a1);
833     }
834
835     // XZ plane, a2 fracture = node 3 -> node 4
836     dE = ((n8*n8+s3*s3+s4*s4)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
837     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
838     if(-(dE/dA) >= gc){
839         fractures[(i+Nx)*6+0] += delta_fracture;
840         a2 += delta_fracture;
841         u3.x = U[i+Nx].x / (1.f+fracture_stress_release*a2);
842     }
843
844     // YZ plane, a1 fracture = node 3 -> node 1
845     a2 = fractures[(i+Nx)*6+2]; // +Y
846     dE = ((n3*n3+s4*s4+s8*s8)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
847     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
848     if(-(dE/dA) >= gc){
849         fractures[(i+Nx)*6+5] += delta_fracture;
850         a1 += delta_fracture;
851         u3.z = U[i+Nx].z / (1.f+fracture_stress_release*a1);
852     }
853
854     // YZ plane, a2 fracture = node 3 -> node 7
855     dE = ((n3*n3+s4*s4+s8*s8)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
856     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
857     if(-(dE/dA) >= gc){
858         fractures[(i+Nx)*6+2] += delta_fracture;
859         a2 += delta_fracture;
860         u3.y = U[i+Nx].y / (1.f+fracture_stress_release*a2);
861     }
862
863     // YX plane, a1 fracture = node 3 -> node 4
864     a1 = fractures[(i+Nx)*6+0]; // +X
865     dE = ((n4*n4+s3*s3+s8*s8)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
866     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
867     if(-(dE/dA) >= gc){
868         fractures[(i+Nx)*6+0] += delta_fracture;
869         a1 += delta_fracture;
870         u3.x = U[i+Nx].x / (1.f+fracture_stress_release*a1);
871     }
872
873     // YX plane, a2 fracture = node 3 -> node 7
```

```
874     dE = ((n4*n4+s3*s3+s8*s8)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
875     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
876     if(-(dE/dA) >= gc){
877         fractures[(i+Nx)*6+2] += delta_fracture;
878         a2 += delta_fracture;
879         u3.y = U[i+Nx].y / (1.f+fracture_stress_release*a2);
880     }
881 
882     // Visualization
883     float temp = (-1.f)*(dE/dA) / gc;
884     if(!isnan(temp)){
885         energy[i+Nx] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
886     }
887 
888     if(!isnan(abs(n3+n4+n8))){
889         normal_stress[i+Nx] = calculate_color(abs(n3+n4+n8), max_normal_stress);
890     }
891 
892     if(!isnan(abs(s3+s4+s8))){
893         shear_stress[i+Nx] = calculate_color(abs(s3+s4+s8), max_shear_stress);
894     }
895 
896     density[i+Nx] = calculate_color(find_density(mesh, U, vertices, i),
            max_density);
897 }
898 
899 /**
900 * Propagating fractures in node 4 corner
901 **/
902 __global__ void propagate_fractures_step4(float3 *vertices, float3 *U, float *
        fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
        shear_stress, float3 *density){
903     // ID
904     int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
905     int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
906     int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
907     int i = id_y*Nx*Nz + id_z*Nx + id_x;
908     float dA, dE, E;
909     float a1, a2;
910     float beta = 3.1415f;
911 
912     // Threads outside domain
913     if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
914         return;
915     }
916 
917     // Young's modulus
918     float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
919     E = find_youngs_modulus(mesh, U, vertices, i);
920 
921     // Local displacement
922     real3 u2, u3, u4, u8;
923     u2.x = U[i+1].x / (1.f+fracture_stress_release*(fractures[(i+1)*6+2]+fractures
            [(i+1)*6+4]));
924     u2.y = U[i+1].y / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+4]));
925     u2.z = U[i+1].z / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
            [(i+1)*6+2]));
926     u3.x = U[i+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+2]+
            fractures[(i+Nx)*6+5]));
927     u3.y = U[i+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
            fractures[(i+Nx)*6+5]));
928     u3.z = U[i+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
            fractures[(i+Nx)*6+2]));
929     u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+2]+
            fractures[(i+Nx+1)*6+5]));
930     u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
            fractures[(i+Nx+1)*6+5]));
931     u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
            fractures[(i+Nx+1)*6+2]));
932     u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
            +1)*6+3]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
933     u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
            +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
```

```
934     u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
            +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+3]));
935
936     // Global vertices
937     real3 v2, v3, v4, v8;
938     v2.x = vertices[i+1].x;
939     v2.y = vertices[i+1].y;
940     v2.z = vertices[i+1].z;
941     v3.x = vertices[i+Nx].x;
942     v3.y = vertices[i+Nx].y;
943     v3.z = vertices[i+Nx].z;
944     v4.x = vertices[i+Nx+1].x;
945     v4.y = vertices[i+Nx+1].y;
946     v4.z = vertices[i+Nx+1].z;
947     v8.x = vertices[i+Nx*Nz+Nx+1].x;
948     v8.y = vertices[i+Nx*Nz+Nx+1].y;
949     v8.z = vertices[i+Nx*Nz+Nx+1].z;
950
951     // Normal strain
952     real n2, n3, n7, theta;
953     theta = dot(u4-u2, v4-v2)/(length(u4-u2) * length(v4-v2));
954     n2  = (length(u4-u2)*theta)/length(v4-v2)*E;
955     theta = dot(u4-u3, v4-v3)/(length(u4-u3) * length(v4-v3));
956     n3  = (length(u4-u3)*theta)/length(v4-v3)*E;
957     theta = dot(u8-u4, v8-v4)/(length(u8-u4) * length(v8-v4));
958     n7  = (length(u8-u4)*theta)/length(v8-v4)*E;
959
960     // Shear strain
961     real s2, s3, s7;
962     s2  = acos(dot(v4-v2, (v4+u4)-(v2+u2))/(length(v4-v2) * length((v4+u4)-(v2+u2)
            )))*E;
963     s3  = acos(dot(v4-v3, (v4+u4)-(v3+u3))/(length(v4-v3) * length((v4+u4)-(v3+u3)
            )))*E;
964     s7  = acos(dot(v8-v4, (v8+u8)-(v4+u4))/(length(v8-v4) * length((v8+u8)-(v4+u4)
            )))*E;
965
966     // XZ plane, a1 fracture = node 4 -> node 3
967     a1 = fractures[(i+Nx+1)*6+1]; // -X
968     a2 = fractures[(i+Nx+1)*6+5]; // -Z
969     dE = ((n7*n7+s2*s2+s3*s3)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
970     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
971     if(-(dE/dA) >= gc){
972         fractures[(i+Nx+1)*6+1] += delta_fracture;
973         a1 += delta_fracture;
974         u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*a1);
975     }
976
977     // XZ plane a2 fracture = node 4 -> node 2
978     dE = ((n7*n7+s2*s2+s3*s3)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
979     dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
980     if(-(dE/dA) >= gc){
981         fractures[(i+Nx+1)*6+5] += delta_fracture;
982         a2 += delta_fracture;
983         u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*a2);
984     }
985
986     // YZ plane, a1 fracture = node 4 -> node 8
987     a1 = fractures[(i+Nx+1)*6+2]; // +Y
988     dE = ((n3*n3+s2*s2+s7*s7)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
            *a2));
989     dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
990     if(-(dE/dA) >= gc){
991         fractures[(i+Nx+1)*6+2] += delta_fracture;
992         a1 += delta_fracture;
993         u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*a1);
994     }
995
996     // YZ plane a2 fracture = node 4 -> node 3
997     dE = ((n3*n3+s2*s2+s7*s7)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
            sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
            delta_fracture)));
```

```
998       dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
999       if(-(dE/dA) >= gc){
1000          fractures[(i+Nx+1)*6+5] += delta_fracture;
1001          a2 += delta_fracture;
1002          u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*a2);
1003      }
1004
1005      // YX plane a1 fracture = node 4 -> node 8
1006      a2 = fractures[(i+Nx+1)*6+1]; // -X
1007      dE = ((n2*n2+s3*s3+s7*s7)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
              *a2));
1008      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1009      if(-(dE/dA) >= gc){
1010          fractures[(i+Nx+1)*6+2] += delta_fracture;
1011          a1 += delta_fracture;
1012          u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*a1);
1013      }
1014
1015      // YX plane a2 fracture = node 4 -> node 3
1016      dE = ((n3*n3+s2*s2+s7*s7)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1017      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1018      if(-(dE/dA) >= gc){
1019          fractures[(i+Nx+1)*6+1] += delta_fracture;
1020          a2 += delta_fracture;
1021          u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*a2);
1022      }
1023
1024      // Visualization
1025      float temp = (-1.f)*(dE/dA) / gc;
1026      if(!isnan(temp)){
1027          energy[i+Nx+1] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
1028      }
1029
1030      if(!isnan(abs(n2+n3+n7))){
1031          normal_stress[i+Nx+1] = calculate_color(abs(n2+n3+n7), max_normal_stress);
1032      }
1033
1034      if(!isnan(abs(s2+s3+s7))){
1035          shear_stress[i+Nx+1] = calculate_color(abs(s2+s3+s7), max_shear_stress);
1036      }
1037
1038      density[i+Nx+1] = calculate_color(find_density(mesh, U, vertices, i),
              max_density);
1039  }
1040
1041  /**
1042  * Propagating fractures in node 5 corner
1043  **/
1044  __global__ void propagate_fractures_step5(float3 *vertices, float3 *U, float *
          fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
          shear_stress, float3 *density){
1045      // ID
1046      int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
1047      int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
1048      int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
1049      int i = id_y*Nx*Nz + id_z*Nx + id_x;
1050      float dA, dE, E;
1051      float a1, a2;
1052      float beta = 3.1415f;
1053
1054      // Threads outside domain
1055      if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
1056          return;
1057      }
1058
1059      // Young's modulus
1060      float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
1061      E = find_youngs_modulus(mesh, U, vertices, i);
1062
1063      // Local displacement
1064      real3 u1, u5, u6, u7;
1065      u1.x = U[i].x / (1.f+fracture_stress_release*(fractures[i*6+2]+fractures[i
              *6+4]));
```

```
1066        u1.y = U[i].y / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
                *6+4]));
1067        u1.z = U[i].z / (1.f+fracture_stress_release*(fractures[i*6+0]+fractures[i
                *6+2]));
1068        u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+3]+
                fractures[(i+Nx*Nz)*6+4]));
1069        u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
                fractures[(i+Nx*Nz)*6+4]));
1070        u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
                fractures[(i+Nx*Nz)*6+3]));
1071        u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
                *6+3]+fractures[(i+Nx*Nz+1)*6+4]));
1072        u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
                *6+1]+fractures[(i+Nx*Nz+1)*6+4]));
1073        u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
                *6+1]+fractures[(i+Nx*Nz+1)*6+3]));
1074        u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+3]+fractures[(i+Nx*Nz+Nx)*6+5]));
1075        u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+5]));
1076        u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+3]));
1077
1078        // Global vertices
1079        real3 v1, v5, v6, v7;
1080        v1.x = vertices[i].x;
1081        v1.y = vertices[i].y;
1082        v1.z = vertices[i].z;
1083        v5.x = vertices[i+Nx*Nz].x;
1084        v5.y = vertices[i+Nx*Nz].y;
1085        v5.z = vertices[i+Nx*Nz].z;
1086        v6.x = vertices[i+Nx*Nx+1].x;
1087        v6.y = vertices[i+Nx*Nx+1].y;
1088        v6.z = vertices[i+Nx*Nx+1].z;
1089        v7.x = vertices[i+Nx*Nz+Nx].x;
1090        v7.y = vertices[i+Nx*Nz+Nx].y;
1091        v7.z = vertices[i+Nx*Nz+Nx].z;
1092
1093        // Normal strain
1094        real n5, n9, n12, theta;
1095        theta = dot(u5-u1, v5-v1)/(length(u5-u1) * length(v5-v1));
1096        n5  = (length(u5-u1)*theta)/length(v5-v1)*E;
1097        theta = dot(u6-u5, v6-v5)/(length(u6-u5) * length(v6-v5));
1098        n9  = (length(u6-u5)*theta)/length(v6-v5)*E;
1099        theta = dot(u7-u5, v7-v5)/(length(u7-u5) * length(v7-v5));
1100        n12 = (length(u7-u5)*theta)/length(v7-v5)*E;
1101
1102        // Shear strain
1103        real s5, s9, s12;
1104        s5  = acos(dot(v5-v1, (v5+u5)-(v1+u1))/(length(v5-v1) * length((v5+u5)-(v1+u1)
                )))*E;
1105        s9  = acos(dot(v6-v5, (v6+u6)-(v5+u5))/(length(v6-v5) * length((v6+u6)-(v5+u5)
                )))*E;
1106        s12 = acos(dot(v7-v5, (v7+u7)-(v5+u5))/(length(v7-v5) * length((v7+u7)-(v5+u5)
                )))*E;
1107
1108        // XZ plane, a1 fracture = node 5 -> node 6
1109        a1 = fractures[(i+Nx*Nz)*6+0]; // +X
1110        a2 = fractures[(i+Nx*Nz)*6+4]; // +Z
1111        dE = ((n5*n5+s9*s9+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
                *a2));
1112        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1113        if(-(dE/dA) >= gc){
1114            fractures[(i+Nx*Nz)*6+0] += delta_fracture;
1115            a1 += delta_fracture;
1116            u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*a1);
1117        }
1118
1119        // XZ plane, a2 fracture = node 5 -> node 7
1120        dE = ((n5*n5+s9*s9+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1121        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1122        if(-(dE/dA) >= gc){
1123            fractures[(i+Nx*Nz)*6+4] += delta_fracture;
```

```
1124          a2 += delta_fracture;
1125          u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*a2);
1126      }
1127
1128      //YZ plane, a1 fracture = node 5 -> node 1
1129      a1 = fractures[(i+Nx*Nz)*6+3]; // -Y
1130      dE = ((n9*n9+s5*s5+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
              *a2));
1131      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1132      if(-(dE/dA) >= gc){
1133          fractures[(i+Nx*Nz)*6+3] += delta_fracture;
1134          a1 += delta_fracture;
1135          u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*a1);
1136      }
1137
1138      // YZ plane, a2 fracture = node 5 -> node 7
1139      dE = ((n9*n9+s5*s5+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1140      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1141      if(-(dE/dA) >= gc){
1142          fractures[(i+Nx*Nz)*6+4] += delta_fracture;
1143          a2 += delta_fracture;
1144          u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*a2);
1145      }
1146
1147      // YX plane, a1 fracture = node 5 -> node 1
1148      a2 = fractures[(i+Nx*Nz)*6+0];
1149      dE = ((n12*n12+s5*s5+s9*s9)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
              *a2));
1150      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1151      if(-(dE/dA) >= gc){
1152          fractures[(i+Nx*Nz)*6+3] += delta_fracture;
1153          a1 += delta_fracture;
1154          u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*a1);
1155      }
1156
1157      // YX plane, a2 fracture = node 5 -> node 6
1158      dE = ((n12*n12+s5*s5+s9*s9)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
              sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1159      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1160      if(-(dE/dA) >= gc){
1161          fractures[(i+Nx*Nz)*6+0] += delta_fracture;
1162          a2 += delta_fracture;
1163          u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*a2);
1164      }
1165
1166      // Visualization
1167      float temp = (-1.f)*(dE/dA) / gc;
1168      if(!isnan(temp)){
1169          energy[i+Nx*Nz] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
1170      }
1171
1172      if(!isnan(abs(n5+n9+n12))){
1173          normal_stress[i+Nx*Nz] = calculate_color(abs(n5+n9+n12), max_normal_stress
                  );
1174      }
1175
1176      if(!isnan(abs(s5+s9+s12))){
1177          shear_stress[i+Nx*Nz] = calculate_color(abs(s5+s9+s12), max_shear_stress);
1178      }
1179
1180      density[i+Nx*Nz] = calculate_color(find_density(mesh, U, vertices, i),
              max_density);
1181  }
1182
1183  /**
1184  * Propagating fractures in node 6 corner
1185  **/
1186  __global__ void propagate_fractures_step6(float3 *vertices, float3 *U, float *
          fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
          shear_stress, float3 *density){
1187      // ID
```

```
1188      int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
1189      int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
1190      int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
1191      int i = id_y*Nx*Nz + id_z*Nx + id_x;
1192      float dA, dE, E;
1193      float a1, a2;
1194      float beta = 3.1415f;
1195
1196      // Threads outside domain
1197      if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
1198          return;
1199      }
1200
1201      // Young's modulus
1202      float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
1203      E = find_youngs_modulus(mesh, U, vertices, i);
1204
1205      // Local displacement
1206      real3 u2, u5, u6, u8;
1207      u2.x = U[i+1].x / (1.f+fracture_stress_release*(fractures[(i+1)*6+2]+fractures
              [(i+1)*6+4]));
1208      u2.y = U[i+1].y / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
              [(i+1)*6+4]));
1209      u2.z = U[i+1].z / (1.f+fracture_stress_release*(fractures[(i+1)*6+1]+fractures
              [(i+1)*6+2]));
1210      u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+3]+
              fractures[(i+Nx*Nz)*6+4]));
1211      u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
              fractures[(i+Nx*Nz)*6+4]));
1212      u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
              fractures[(i+Nx*Nz)*6+3]));
1213      u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+1]+fractures[(i+Nx*Nz+1)*6+4]));
1214      u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+1]+fractures[(i+Nx*Nz+1)*6+4]));
1215      u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+1]+fractures[(i+Nx*Nz+1)*6+3]));
1216      u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+3]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
1217      u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
1218      u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+3]));
1219
1220      // Global vertices
1221      real3 v2, v5, v6, v8;
1222      v2.x = vertices[i+1].x;
1223      v2.y = vertices[i+1].y;
1224      v2.z = vertices[i+1].z;
1225      v5.x = vertices[i+Nx*Nz].x;
1226      v5.y = vertices[i+Nx*Nz].y;
1227      v5.z = vertices[i+Nx*Nz].z;
1228      v6.x = vertices[i+Nx*Nx+1].x;
1229      v6.y = vertices[i+Nx*Nx+1].y;
1230      v6.z = vertices[i+Nx*Nx+1].z;
1231      v8.x = vertices[i+Nx*Nz+Nx+1].x;
1232      v8.y = vertices[i+Nx*Nz+Nx+1].y;
1233      v8.z = vertices[i+Nx*Nz+Nx+1].z;
1234
1235      // Normal strain
1236      real n6, n9, n10, theta;
1237      theta = dot(u6-u2, v6-v2)/(length(u6-u2) * length(v6-v2));
1238      n6  = (length(u6-u2)*theta)/length(v6-v2)*E;
1239      theta = dot(u6-u5, v6-v5)/(length(u6-u5) * length(v6-v5));
1240      n9  = (length(u6-u5)*theta)/length(v6-v5)*E;
1241      theta = dot(u8-u6, v8-v6)/(length(u8-u6) * length(v8-v6));
1242      n10 = (length(u8-u6)*theta)/length(v8-v6)*E;
1243
1244      // Shear strain
1245      real s6, s9, s10;
1246      s6  = acos(dot(v6-v2, (v6+u6)-(v2+u2))/(length(v6-v2) * length((v6+u6)-(v2+u2)
              )))*E;
1247      s9  = acos(dot(v6-v5, (v6+u6)-(v5+u5))/(length(v6-v5) * length((v6+u6)-(v5+u5)
              )))*E;
1248      s10 = acos(dot(v8-v6, (v8+u8)-(v6+u6))/(length(v8-v6) * length((v8+u8)-(v6+u6)
              )))*E;
```

```
1249
1250        // XZ plane, a1 fracture = node 6 -> node 5
1251        a1 = fractures[(i+Nx*Nz+1)*6+1]; // -X
1252        a2 = fractures[(i+Nx*Nz+1)*6+4]; // +Z
1253        dE = ((n6*n6+s9*s9+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
                *a2));
1254        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1255        if(-(dE/dA) >= gc){
1256            fractures[(i+Nx*Nz+1)*6+1] += delta_fracture;
1257            a1 += delta_fracture;
1258            u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*a1);
1259        }
1260
1261        // XZ plane, a2 fracture = node 6 -> node 8
1262        dE = ((n6*n6+s9*s9+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1263        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1264        if(-(dE/dA) >= gc){
1265            fractures[(i+Nx*Nz+1)*6+4] += delta_fracture;
1266            a2 += delta_fracture;
1267            u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*a2);
1268        }
1269
1270        // YZ plane, a1 fracture = node 6 -> node 2
1271        a1 = fractures[(i+Nx*Nz+1)*6+3]; // -Y
1272        dE = ((n9*n9+s6*s6+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
                *a2));
1273        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1274        if(-(dE/dA) >= gc){
1275            fractures[(i+Nx*Nz+1)*6+3] += delta_fracture;
1276            a1 += delta_fracture;
1277            u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*a1);
1278        }
1279
1280        // YZ plane, a2 fracture = node 6 -> node 8
1281        dE = ((n9*n9+s6*n6+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1282        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1283        if(-(dE/dA) >= gc){
1284            fractures[(i+Nx*Nz+1)*6+4] += delta_fracture;
1285            a2 += delta_fracture;
1286            u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*a2);
1287        }
1288
1289        // YX plane, a1 fracture = node 6 -> node 2
1290        a2 = fractures[(i+Nx*Nz+1)*6+1]; // -X
1291        dE = ((n10*n10+s6*s6+s9*s9)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+delta_fracture)
                *a2));
1292        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1293        if(-(dE/dA) >= gc){
1294            fractures[(i+Nx*Nz+1)*6+3] += delta_fracture;
1295            a1 += delta_fracture;
1296            u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*a1);
1297        }
1298
1299        // YX plane, a2 fracture = node 6 -> node 5
1300        dE = ((n10*n10+s6*s6+s9*s9)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((beta*
                sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1301        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1302        if(-(dE/dA) >= gc){
1303            fractures[(i+Nx*Nz+1)*6+1] += delta_fracture;
1304            a2 += delta_fracture;
1305            u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*a2);
1306        }
1307
1308        // Visualization
1309        float temp = (-1.f)*(dE/dA) / gc;
1310        if(!isnan(temp)){
1311            energy[i+Nx*Nz+1] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
1312        }
```

```
1313
1314        if(!isnan(abs(n6+n9+n10))){
1315            normal_stress[i+Nx*Nz+1] = calculate_color(abs(n6+n9+n10),
                    max_normal_stress);
1316        }
1317
1318        if(!isnan(abs(s6+s9+s10))){
1319            shear_stress[i+Nx*Nz+1] = calculate_color(abs(s6+s9+s10), max_shear_stress
                    );
1320        }
1321
1322        density[i+Nx*Nz+1] = calculate_color(find_density(mesh, U, vertices, i),
                max_density);
1323    }
1324
1325    /**
1326    * Propagating fractures in node 7 corner
1327    **/
1328    __global__ void propagate_fractures_step7(float3 *vertices, float3 *U, float *
            fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
            shear_stress, float3 *density){
1329        // ID
1330        int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
1331        int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
1332        int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
1333        int i = id_y*Nx*Nz + id_z*Nx + id_x;
1334        float dA, dE, E;
1335        float a1, a2;
1336        float beta = 3.1415f;
1337
1338        // Threads outside domain
1339        if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
1340            return;
1341        }
1342
1343        // Young's modulus
1344        float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
1345        E = find_youngs_modulus(mesh, U, vertices, i);
1346
1347        // Local displacement
1348        real3 u3, u5, u7, u8;
1349        u3.x = U[i+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+2]+
                fractures[(i+Nx)*6+5]));
1350        u3.y = U[i+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
                fractures[(i+Nx)*6+5]));
1351        u3.z = U[i+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx)*6+0]+
                fractures[(i+Nx)*6+2]));
1352        u5.x = U[i+Nx*Nz].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+3]+
                fractures[(i+Nx*Nz)*6+4]));
1353        u5.y = U[i+Nx*Nz].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
                fractures[(i+Nx*Nz)*6+4]));
1354        u5.z = U[i+Nx*Nz].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz)*6+0]+
                fractures[(i+Nx*Nz)*6+3]));
1355        u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+3]+fractures[(i+Nx*Nz+Nx)*6+5]));
1356        u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+5]));
1357        u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
                *6+0]+fractures[(i+Nx*Nz+Nx)*6+3]));
1358        u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*fractures[(i+Nx*Nz+Nx
                +1)*6+1]);
1359        u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*fractures[(i+Nx*Nz+Nx
                +1)*6+3]);
1360        u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*fractures[(i+Nx*Nz+Nx
                +1)*6+5]);
1361
1362        // Global vertices
1363        real3 v3, v5, v7, v8;
1364        v3.x = vertices[i+Nx].x;
1365        v3.y = vertices[i+Nx].y;
1366        v3.z = vertices[i+Nx].z;
1367        v5.x = vertices[i+Nx*Nz].x;
1368        v5.y = vertices[i+Nx*Nz].y;
1369        v5.z = vertices[i+Nx*Nz].z;
1370        v7.x = vertices[i+Nx*Nz+Nx].x;
1371        v7.y = vertices[i+Nx*Nz+Nx].y;
```

```
1372        v7.z = vertices[i+Nx*Nz+Nx].z;
1373        v8.x = vertices[i+Nx*Nz+Nx+1].x;
1374        v8.y = vertices[i+Nx*Nz+Nx+1].y;
1375        v8.z = vertices[i+Nx*Nz+Nx+1].z;
1376
1377        // Normal strain
1378        real n8, n11, n12, theta;
1379        theta = dot(u7-u3, v7-v3)/(length(u7-u3) * length(v7-v3));
1380        n8  = (length(u7-u3)*theta)/length(v7-v3)*E;
1381        theta = dot(u8-u7, v8-v7)/(length(u8-u7) * length(v8-v7));
1382        n11 = (length(u8-u7)*theta)/length(v8-v7)*E;
1383        theta = dot(u7-u5, v7-v5)/(length(u7-u5) * length(v7-v5));
1384        n12 = (length(u7-u5)*theta)/length(v7-v5)*E;
1385
1386        // Shear strain
1387        real s8, s11, s12;
1388        s8  = acos(dot(v7-v3, (v7+u7)-(v3+u3))/(length(v7-v3) * length((v7+u7)-(v3+u3)
                ))*E;
1389        s11 = acos(dot(v8-v7, (v8+u8)-(v7+u7))/(length(v8-v7) * length((v8+u8)-(v7+u7)
                ))*E;
1390        s12 = acos(dot(v7-v5, (v7+u7)-(v5+u5))/(length(v7-v5) * length((v7+u7)-(v5+u5)
                ))*E;
1391
1392        // XZ plane, a1 fracture = node 7 -> node 8
1393        a1 = fractures[(i+Nx*Nz+Nx)*6+0]; // +X
1394        a2 = fractures[(i+Nx*Nz+Nx)*6+5]; // -Z
1395        dE = ((n8*n8+s11*s11+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
                beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
                delta_fracture)*a2));
1396        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1397        if(-(dE/dA) >= gc){
1398            fractures[(i+Nx*Nz+Nx)*6+0] += delta_fracture;
1399            a1 += delta_fracture;
1400            u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*a1);
1401        }
1402
1403        // XZ plane, a2 fracture = node 7 -> node 5
1404        dE = ((n8*n8+s11*s11+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
                beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1405        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1406        if(-(dE/dA) >= gc){
1407            fractures[(i+Nx*Nz+Nx)*6+5] += delta_fracture;
1408            a2 += delta_fracture;
1409            u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*a2);
1410        }
1411
1412        // YZ plane, a1 fracture = node 7 -> node 3
1413        a1 = fractures[(i+Nx*Nz+Nx)*6+3]; // -Y
1414        dE = ((n11*n11+s8*s8+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
                beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
                delta_fracture)*a2));
1415        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1416        if(-(dE/dA) >= gc){
1417            fractures[(i+Nx*Nz+Nx)*6+3] += delta_fracture;
1418            a1 += delta_fracture;
1419            u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*a1);
1420        }
1421
1422        // YZ plane, a2 fracture = node 7 -> node 5
1423        dE = ((n11*n11+s8*s8+s12*s12)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
                beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
                delta_fracture)));
1424        dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1425        if(-(dE/dA) >= gc){
1426            fractures[(i+Nx*Nz+Nx)*6+5] += delta_fracture;
1427            a2 += delta_fracture;
1428            u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*a2);
1429        }
1430
1431        // YX plane, a1 fracture = node 7 -> node 3
1432        a2 = fractures[(i+Nx*Nz+Nx)*6+0]; // +X
1433        dE = ((n12*n12+s8*s8+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
                beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
                delta_fracture)*a2));
1434        dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
```

```
1435      if(-(dE/dA) >= gc){
1436          fractures[(i+Nx*Nz+Nx)*6+3] += delta_fracture;
1437          a1 += delta_fracture;
1438          u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*a1);
1439      }
1440
1441      // YX plane, a2 fracture = node 7 -> node 8
1442      dE = ((n12*n12+s8*s8+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1443      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1444      if(-(dE/dA) >= gc){
1445          fractures[(i+Nx*Nz+Nx)*6+0] += delta_fracture;
1446          a2 += delta_fracture;
1447          u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*a2);
1448      }
1449
1450      // Visualization
1451      float temp = (-1.f)*(dE/dA) / gc;
1452      if(!isnan(temp)){
1453          energy[i+Nx*Nz+Nx] = calculate_color(((-1.f)*(dE/dA) / gc), 1.f);
1454      }
1455
1456      if(!isnan(abs(n8+n11+n12))){
1457          normal_stress[i+Nx*Nz+Nx] = calculate_color(abs(n8+n11+n12),
                  max_normal_stress);
1458      }
1459
1460      if(!isnan(abs(s8+s11+s12))){
1461          shear_stress[i+Nx*Nz+Nx] = calculate_color(abs(s8+s11+s12),
                  max_shear_stress);
1462      }
1463
1464      density[i+Nx*Nz+Nx] = calculate_color(find_density(mesh, U, vertices, i),
              max_density);
1465  }
1466
1467  /**
1468  * Propagating fractures in node 8 corner
1469  **/
1470  __global__ void propagate_fractures_step8(float3 *vertices, float3 *U, float *
          fractures, mesh_point *mesh, float3 *energy, float3 *normal_stress, float3 *
          shear_stress, float3 *density){
1471      // ID
1472      int id_x = (blockIdx.x * blockDim.x) + threadIdx.x;
1473      int id_y = (blockIdx.y * blockDim.y) + threadIdx.y;
1474      int id_z = (blockIdx.z * blockDim.z) + threadIdx.z;
1475      int i = id_y*Nx*Nz + id_z*Nx + id_x;
1476      float dA, dE, E;
1477      float a1, a2;
1478      float beta = 3.1415f;
1479
1480      // Threads outside domain
1481      if(id_x >= Ex || id_y >= Ey || id_z >= Ez){
1482          return;
1483      }
1484
1485      // Young's modulus
1486      float gc = find_critical_energy_release_rate(mesh, U, vertices, i);
1487      E = find_youngs_modulus(mesh, U, vertices, i);
1488
1489      // Local displacement
1490      real3 u4, u6, u7, u8;
1491      u4.x = U[i+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+2]+
              fractures[(i+Nx+1)*6+5]));
1492      u4.y = U[i+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
              fractures[(i+Nx+1)*6+5]));
1493      u4.z = U[i+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx+1)*6+1]+
              fractures[(i+Nx+1)*6+2]));
1494      u6.x = U[i+Nx*Nz+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+3]+fractures[(i+Nx*Nz+1)*6+4]));
1495      u6.y = U[i+Nx*Nz+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+1]+fractures[(i+Nx*Nz+1)*6+4]));
1496      u6.z = U[i+Nx*Nz+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+1)
              *6+1]+fractures[(i+Nx*Nz+1)*6+3]));
```

```
1497      u7.x = U[i+Nx*Nz+Nx].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
              *6+3]+fractures[(i+Nx*Nz+Nx)*6+5]));
1498      u7.y = U[i+Nx*Nz+Nx].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
              *6+0]+fractures[(i+Nx*Nz+Nx)*6+5]));
1499      u7.z = U[i+Nx*Nz+Nx].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx)
              *6+0]+fractures[(i+Nx*Nz+Nx)*6+3]));
1500      u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+3]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
1501      u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+5]));
1502      u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*(fractures[(i+Nx*Nz+Nx
              +1)*6+1]+fractures[(i+Nx*Nz+Nx+1)*6+3]));
1503
1504      // Global vertices
1505      real3 v4, v6, v7, v8;
1506      v4.x = vertices[i+Nx+1].x;
1507      v4.y = vertices[i+Nx+1].y;
1508      v4.z = vertices[i+Nx+1].z;
1509      v6.x = vertices[i+Nx*Nx+1].x;
1510      v6.y = vertices[i+Nx*Nx+1].y;
1511      v6.z = vertices[i+Nx*Nx+1].z;
1512      v7.x = vertices[i+Nx*Nz+Nx].x;
1513      v7.y = vertices[i+Nx*Nz+Nx].y;
1514      v7.z = vertices[i+Nx*Nz+Nx].z;
1515      v8.x = vertices[i+Nx*Nz+Nx+1].x;
1516      v8.y = vertices[i+Nx*Nz+Nx+1].y;
1517      v8.z = vertices[i+Nx*Nz+Nx+1].z;
1518
1519      // Normal strain
1520      real n7, n10, n11, theta;
1521      theta = dot(u8-u4, v8-v4)/(length(u8-u4) * length(v8-v4));
1522      n7  = (length((u8-u4))*theta)/length(v8-v4)*E;
1523      theta = dot(u8-u6, v8-v6)/(length(u8-u6) * length(v8-v6));
1524      n10 = (length((u8-u6))*theta)/length(v8-v6)*E;
1525      theta = dot(u8-u7, v8-v7)/(length(u8-u7) * length(v8-v7));
1526      n11 = (length((u8-u7))*theta)/length(v8-v7)*E;
1527
1528      // Shear strain
1529      real s7, s10, s11;
1530      s7  = acos(dot(v8-v4, (v8+u8)-(v4+u4))/(length(v8-v4) * length((v8+u8)-(v4+u4)
              )))*E;
1531      s10 = acos(dot(v8-v6, (v8+u8)-(v6+u6))/(length(v8-v6) * length((v8+u8)-(v6+u6)
              )))*E;
1532      s11 = acos(dot(v8-v7, (v8+u8)-(v7+u7))/(length(v8-v7) * length((v8+u8)-(v7+u7)
              )))*E;
1533
1534      // XZ plane, a1 fracture = node 8 -> node 7
1535      a1 = fractures[(i+Nx*Nz+Nx+1)*6+1]; // -X
1536      a2 = fractures[(i+Nx*Nz+Nx+1)*6+5]; // -Z
1537      dE = ((n7*n7+s10*s10+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
              delta_fracture)*a2));
1538      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1539      if(-(dE/dA) >= gc){
1540          fractures[(i+Nx*Nz+Nx+1)*6+1] += delta_fracture;
1541          a1 += delta_fracture;
1542          u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*a1);
1543      }
1544
1545      // XZ plane, a2 fracture = node 8 -> node 6
1546      dE = ((n7*n7+s10*s10+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1547      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1548      if(-(dE/dA) >= gc){
1549          fractures[(i+Nx*Nz+Nx+1)*6+5] += delta_fracture;
1550          a2 += delta_fracture;
1551          u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*a2);
1552      }
1553
1554      // YZ plane, a1 fracture = node 8 -> node 4
1555      a1 = fractures[(i+Nx*Nz+Nx+1)*6+3]; // -Y
1556      dE = ((n11*n11+s7*s7+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
              delta_fracture)*a2));
1557      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
```

```
1558      if(-(dE/dA) >= gc){
1559          fractures[(i+Nx*Nz+Nx+1)*6+3] += delta_fracture;
1560          a1 += delta_fracture;
1561          u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*a1);
1562      }
1563
1564      // YZ plane, a2 fracture = node 8 -> node 6
1565      dE = ((n11*n11+s7*s7+s10*s10)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1566      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1567      if(-(dE/dA) >= gc){
1568          fractures[(i+Nx*Nz+Nx+1)*6+5] += delta_fracture;
1569          a2 += delta_fracture;
1570          u8.z = U[i+Nx*Nz+Nx+1].z / (1.f+fracture_stress_release*a2);
1571      }
1572
1573      // YX plane, a1 fracture = node 8 -> node 4
1574      a2 = fractures[(i+Nx*Nz+Nx+1)*6+1]; // -X
1575      dE = ((n10*n10+s7*s7+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt((a1+delta_fracture)*(a1+delta_fracture) + a2*a2))*(a1+
              delta_fracture)*a2));
1576      dA = (((a1+delta_fracture)*a2) - (a1*a2))/2.f;
1577      if(-(dE/dA) >= gc){
1578          fractures[(i+Nx*Nz+Nx+1)*6+3] += delta_fracture;
1579          a1 += delta_fracture;
1580          u8.y = U[i+Nx*Nz+Nx+1].y / (1.f+fracture_stress_release*a1);
1581      }
1582
1583      // YX plane, a2 fracture = node 8 -> node 7
1584      dE = ((n10*n10+s7*s7+s11*s11)/E) * (((beta*sqrt(a1*a1 + a2*a2))*a1*a2) - ((
              beta*sqrt(a1*a1 + (a2+delta_fracture)*(a2+delta_fracture)))*a1*(a2+
              delta_fracture)));
1585      dA = ((a1*(a2+delta_fracture)) - (a1*a2))/2.f;
1586      if(-(dE/dA) >= gc){
1587          fractures[(i+Nx*Nz+Nx+1)*6+1] += delta_fracture;
1588          a2 += delta_fracture;
1589          u8.x = U[i+Nx*Nz+Nx+1].x / (1.f+fracture_stress_release*a2);
1590      }
1591
1592      // Visualization
1593      float temp = (-1.f)*(dE/dA) / gc;
1594      if(!isnan(temp)){
1595          energy[i+Nx*Nz+Nx+1] = calculate_color(temp, 1.f);
1596      }
1597
1598      if(!isnan(abs(n7+n10+n11))){
1599          normal_stress[i+Nx*Nz+Nx+1] = calculate_color(abs(n7+n10+n11),
                  max_normal_stress);
1600      }
1601
1602      if(!isnan(abs(s7+s10+s11))){
1603          shear_stress[i+Nx*Nz+Nx+1] = calculate_color(abs(s7+s10+s11),
                  max_shear_stress);
1604      }
1605
1606      density[i+Nx*Nz+Nx+1] = calculate_color(find_density(mesh, U, vertices, i),
              max_density);
1607  }
```

Bibliography

[1] Kjetil Babington. Terrain rendering techniques for the hpc-lab snow simulator. Master's thesis, Norwegian University of Science and Technology, 2012.

[2] Robin Eidissen. Utilizing gpus for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology, 2009.

[3] WSL Institute for Snow and Avalanche Research SLF. `http://www.slf.ch/ueber/organisation/schnee_permafrost/projekte/snowpack/index_EN`.

[4] A. A. Griffith. The phenomena of rupture and flow in solids. Technical report, Philosophical Transactions, 1920.

[5] Hiroyuki Hirashima, Kouichi Nishimura, Satoru Yamaguchi, Atsushi Sato, and Michael Lehning. Avalanche forecasting in a heavy snowfall area using the snowpack model. Technical report, Cold Regions Science and Technology, 2008.

[6] David V. Hutton. *Fundamentals of Finite Element Analysis*. The McGraw-Hill Companies, 2004.

[7] C. E. Inglis. Stresses in a plate due to the presence of cracks and sharp corners. Technical report, Transactions of the Institute of Naval Architects, 1913.

[8] G. R. Irwin. Fracturing of metals. Technical report, American Society for Metals, 1948.

[9] G. R. Irwin. Onset of fast crack propagation in high strength steel and aluminum alloys. Technical report, NAVAL RESEARCH LABORATORY, 1956.

[10] Michael Lehning, Perry Bartelt, Bob Brown, and Charles Fierz. A physical snowpack model for the swiss avalanche warning: Part iii: meteorological forcing, thin layer formation and evaluation. Technical report, Cold Regions Science and Technology, 2002.

[11] Michael Lehning, Perry Bartelt, Bob Brown, Charles Fierz, and Pramod Satyawali. A physical snowpack model for the swiss avalanche warning: Part ii. snow microstructure. Technical report, Cold Regions Science and Technology, 2002.

[12] Kenneth G. Libbrecht. `www.snowcrystals.com`.

[13] Hallgeir Lien. Procedural generation of roads for use in the snow simulator. Master's thesis, Norwegian University of Science and Technology, 2011.

[14] Magnus Alvestad Mikalsen. Openacc-based snow simulation. Master's thesis, Norwegian University of Science and Technology, 2013.

[15] Eirik Myklebost. The evolution and current state of cuda gpgpu. Technical report, NTNU, December 2013.

[16] Andreas Nordahl. Enhancing the hpc-lab snow simulator with more realistic terrains and other interactive features. Master's thesis, Norwegian University of Science and Technology, 2013.

[17] Michael Lehning Perry Bartelt. A physical snowpack model for the swiss avalanche warning: Part i: numerical model. Technical report, Cold Regions Science and Technology, 2002.

[18] Sirpa Rasmus, Tiia Grönholm, Michaeal Lehning, Kai Rasmus, and Markku Kulmala. Validation of the snowpack model in five different zones in finland. Technical report, Boreal Environment Research, 2007.

[19] David Roylance. Introduction to fracture mechanics. Technical report, Massachusetts Institute of Technology, 2001.

[20] Ingar Saltvik. Parallel methods for real-timevisualization of snow. Master's thesis, Norwegian University of Science and Technology, 2006.

[21] Christian Sigrist. *Measurement of Fracture Mechanical Properties of Snow and Application to Dry Snow Slab Avalanche Release*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2006.

[22] Martin Stoffel. Numerical modelling of snow using finite elements. Technical report, 2005.

[23] Ph.D. T.L. Anderson. *Fracture Mechanics: Fundamentals and Applications*. Taylor & Francis Group, LLC, 6000 Broken Sound Parkway NW, Suite 300, 2005.

[24] Frederik Magnus Johansen Vestre. Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms. Master's thesis, Norwegian University of Science and Technology, 2012.

[25] Wikipedia. `http://en.wikipedia.org/wiki/Loose_snow_avalanche`.

[26] Øivind L. Boge. Snow layer modeling and avalanche prediction for the hpc-lab snow simulator. Technical report, NTNU, December 2013.

[27] Øystein Eklund Krog. Gpu-based real-time snow avalanche simulations. Master's thesis, Norwegian University of Science and Technology, 2010.