Terramechanics based wheel-soil model in a computer game enviroment

Zordix AB

Viktor Knutsson
vikn0004@student.umu.se

Master thesis 30hp

Examiner: Martin Servin (martin.servin@umu.se)
Supervisor: Jens Walker (jens.walker@zordix.com)
Abstract

This thesis aimed to develop deformable a virtual terrain which a vehicle can move in and interact with in a realistic manner. The theory used to calculate how the terrain influences the vehicle is based on terramechanics. The terrain is divided into two separate parts, one for visualization and one for physical collisions. Deformations of the graphical layer is calculated on the GPU using compute shader programming.

The result of the thesis include a tech demo with a small landscape where an alternate terrain vehicle can deform the terrain as it moves around. The method for deforming the graphical layer is made in such a way so that the computational time does not increase as the size of the terrain does, making the method applicable to actual games.
Sammanfattning


Resultatet av detta arbete inkluderar ett tech demo med ett litet landskap där en fyrhjuling kan deformera terrängen när den rör sig. Medoden för deoration av det grafiska lagret är gjort på ett sådant sätt att uträkningstiden inte ökar när storleken på terrängen gör det, vilket betyder att denna metod gör att använda i faktiska spel.
Nomenclature

Soil parameters and variables

\( p \) \hspace{0.5cm} [\text{Pa}] \hspace{0.5cm} \text{Soil pressure}
\( c \) \hspace{0.5cm} [\text{Pa}] \hspace{0.5cm} \text{Cohesion}
\( k'_c \) \hspace{0.5cm} [-] \hspace{0.5cm} \text{Cohesion coefficient}
\( k'_\phi \) \hspace{0.5cm} [-] \hspace{0.5cm} \text{Frictional coefficient}
\( n \) \hspace{0.5cm} [-] \hspace{0.5cm} \text{Deformation exponent}
\( \tau \) \hspace{0.5cm} [\text{Pa}] \hspace{0.5cm} \text{Shear stress}
\( k \) \hspace{0.5cm} [\text{m}] \hspace{0.5cm} \text{Shear deformation modulus}
\( j \) \hspace{0.5cm} [\text{m}] \hspace{0.5cm} \text{Soil displacement}
\( \phi \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Friction angle}
\( \sigma \) \hspace{0.5cm} [\text{Pa}] \hspace{0.5cm} \text{Normal stress}

Vehicle parameters and variables

\( b \) \hspace{0.5cm} [\text{m}] \hspace{0.5cm} \text{Wheel width}
\( r \) \hspace{0.5cm} [\text{m}] \hspace{0.5cm} \text{Wheel radius}
\( s \) \hspace{0.5cm} [-] \hspace{0.5cm} \text{Slip}
\( \beta \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Slip angle}
\( \omega \) \hspace{0.5cm} [\text{Rad/s}] \hspace{0.5cm} \text{Rotational velocity}
\( v_i \) \hspace{0.5cm} [\text{m/s}] \hspace{0.5cm} \text{Speed in } i\text{-direction, } i = x, y, z
\( v_j \) \hspace{0.5cm} [\text{m/s}] \hspace{0.5cm} \text{Slip velocity}
\( \theta \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Angular position of wheel element}
\( \theta_f \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Entry angle of wheel contact patch}
\( \theta_r \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Exit angle of wheel contact patch}
\( \theta_m \) \hspace{0.5cm} [\text{Rad}] \hspace{0.5cm} \text{Maximum stress angle}
\( F_x \) \hspace{0.5cm} [\text{N}] \hspace{0.5cm} \text{Drawbar pull, aka longitudinal force}
\( F_y \) \hspace{0.5cm} [\text{N}] \hspace{0.5cm} \text{Lateral force}
\( F_z \) \hspace{0.5cm} [\text{N}] \hspace{0.5cm} \text{Vertical force}
\( T_y \) \hspace{0.5cm} [\text{N}] \hspace{0.5cm} \text{Torque around lateral axis of wheel}
# Contents

1 Introduction .......................... 1
   1.1 Background .......................... 1
   1.2 Purpose ................................ 1
   1.3 Goal .................................. 1
   1.4 Limitations ............................ 2

2 Theory ................................ 3
   2.1 Wheel-soil interaction model ............... 3
      2.1.1 Slip and slip angle .................. 4
      2.1.2 Normal stress ......................... 4
      2.1.3 Shear stress ........................ 6
      2.1.4 Drawbar pull ........................ 7
      2.1.5 Lateral force ......................... 7
      2.1.6 Vertical force ....................... 7
      2.1.7 Lateral torque ...................... 7
   2.2 Computer graphics ....................... 7
      2.2.1 Graphics pipeline .................... 8
      2.2.2 Shaders ............................. 8
      2.2.3 General-purpose computing on graphics processing units 9
      2.2.4 Textures ............................ 9
   2.3 Numerical integration and algorithms ....... 11
      2.3.1 Trapzoidal method .................... 11
      2.3.2 Perlin noise ........................ 11
      2.3.3 Grid Interpolation ................... 11

3 The tech demo ........................ 13
   3.1 The vehicle .......................... 13
   3.2 The terrain .......................... 14
   3.3 Physics calculation ...................... 14
   3.4 Graphical terrain deformations ............ 15
      3.4.1 The tiles .......................... 15
      3.4.2 The compute shader .................... 18
   3.5 Physical terrain deformations ............. 19

4 Result ................................ 20

5 Extensions and improvements ............... 22
   5.1 Multipass effect ........................ 22
   5.2 Custom wheel collider and vehicle physics .... 22
   5.3 Custom heightfield ....................... 23
   5.4 Deformable areas ........................ 23

6 Discussion ................................ 24

7 Conclusions ............................ 25
A Appendix - Code
A.1 Script - deformTerrain .......................... 26
A.2 Script - tileController .......................... 43
A.3 Script - Density ................................. 59
A.4 Compute shader - customTerrainDeformer .......... 60
1 Introduction

1.1 Background

Zordix started out as a company that developed games for smartphones. The genre for these games were racing games with offroad vehicles, for example snowmobiles and jet-skis. The games became a success which gave Zordix the ability to expand. Now the company focuses on developing games in the same franchise but instead for the current generation of gaming consoles and computers. These machines have much higher performance capabilities than a smartphone which raises the customers expectations of the games quality. One of the ways Zordix will increase the gaming experience is by implementing more advanced vehicle physics and interactable terrain.

Zordix uses a cross-platform game engine called Unity. Unity is a program that provides several tools without which game development at this level would be impossible for a small company like Zordix. The game engine includes a basic physics engine which handles things like object collisions and lighting models. All interactable objects in a game developed in Unity are represented as Unity-objects. These objects can be for example a car in a race simulator game. Creating a custom made game engine would simply be too big of an investment for a small company. The main drawback of using a free game engine like Unity is that the developers do not have access to all the tools and classes used.

1.2 Purpose

The purpose of this thesis is to examine the possibility and limitations of implementing a vehicle-terrain interaction model that satisfies the time limitations required for a game developed for the current generation of consoles, which includes Xbox One, Playstation 4 and Wii U as well as computers with hardware comparable to these machines.

1.3 Goal

The primary goal of this project is to develop a tech demo of a game where a vehicle is moving in a small landscape with dynamically deformable terrain. The vehicle should interact with the terrain in a realistic manner.

The game has framerate requirement of 60 frames per second which is approximately 16 ms per frame. The other parts of the game is approximated to take 10-12 ms in the worst case scenario. This means that the tech demo should be implemented in such a way so that the underlying theory allows for realtime execution below 5 ms per frame. However, this requirement does not apply to the implementation since this will be subject to optimization by professionals at this area, as long as the implementation with certainty can be shown to be optimized below 5 ms per frame. This allows for the focus of the thesis to be directed towards the development of the algorithm.
1.4 Limitations

The model is based on vehicle-soil interaction, more specifically wheel-soil interaction. This means the only object that can interact with the dynamic terrain are wheel colliders, which provides data for the collisions with the terrain such as slip and rotational speed of the wheel.

The tech demo developed has a simple landscape with a single texture. The developed algorithm would currently not work on a bigger landscape with multiple textures.

The model used is made for a rigid wheel on soft terrain, this assumption is valid as long as the terrain is much softer than the wheel.
2 Theory

In this section the vehicle-soil interaction model used will be derived from physical concepts and empirical observations published in other reports. Furthermore, a brief description of the theory behind computer graphics will be given. Using this background theory it will be explained how the graphical processing unit can be used for calculations other than graphical computations.

2.1 Wheel-soil interaction model

The problem of how a vehicle interacts with a relatively soft terrain is generally very complex because both the tyres and the terrain are deformable. However the model used in this thesis assumes that the tyre is rigid and the terrain deformable, this assumption is valid as long as the terrain is much softer than the tyre because then most of the deformations occur in the terrain. The method described in this section is mainly based on three semi-empirical models which estimates the normal and shear stress on the wheel. The first is the Reece-Bekker equation [1]:

\[ p = (c k_c + b \gamma s) \frac{z^n}{b^n}, \]

where \( p \) is the pressure on the wheel from the terrain, \( z \) is the vertical sinkage, \( c \) is the cohesion of the terrain, \( k_c \) and \( k' \) are coefficients of cohesion and friction angle respectively, \( b \) is the width of the wheel and \( n \) is the deformation exponent. This equation was originally derived by Bekker where he used plates and measured the sinkage into terrain of different materials for different pressures. It was later modified by Reece in order to make the model more general which resulted in equation 1. This model is originally used to describe the pressure, \( p \), from the soil on a rectangular plate sunk into the soil a distance \( z \). In this thesis the wheel is modelled as a collection of infinitimally thin rectangular plates each with its own pressure \( p \) which results in the normal stress \( \sigma \).

The second model is the equation relating shear stress with soil displacement proposed by Janosi and Hanamoto [1]:

\[ \tau = \tau_{max} (1 - e^{-j/k}), \]

where \( \tau_{max} \) is the maximum shear stress the terrain can produce, \( k \) is the shear deformation modulus which has the physical quantity of pressure and \( j \) is the soil shear displacement, which is the absolute distance a specific soil parcel has moved from its original position because of the wheel spin.

The last model is the Mohr-Coulomb failure criterion [5] which gives \( \tau_{max} \):

\[ \tau_{max} = c + \sigma \tan \phi, \]

where \( \sigma \) is the normal stress and \( \phi \) is the internal friction angle of the soil.

With these models as a base, expressions for the normal and shear stresses from the soil on the wheel will be derived. These stresses enables derivation of relevant physical properties which governs the vehicle dynamics.
2.1.1 Slip and slip angle

The slip of the wheel is defined as the excess rotational velocity it has compared to a perfectly rolling wheel. Slip is defined by the following equation:

\[ s = \begin{cases} \frac{r\omega - v_x}{r\omega} & |r\omega| > |v_x|, \\ \frac{r\omega - v_x}{v_x} & |r\omega| < |v_x|, \end{cases} \]

where \( s \) is the slip, \( r \) is the radius of the wheel, \( \omega \) is the rotational velocity, \( v_x \) is the velocity component in the direction the wheel is rolling in, see figure 1.

\[ \text{Figure 1: Definition of cartesian coordinates of the wheel. The z-axis is orthogonal to the contact point beneath the wheel. The x-axis is the rolling direction of the wheel, orthogonal to z. The y-axis is the sideways direction of the wheel, governed by the right hand rule.} \]

Slip angle is the angle between the direction of travel and the direction the wheel is pointing in. It is defined by the equation:

\[ \beta = \arctan \frac{v_y}{v_x} \]  

(4)

2.1.2 Normal stress

The normal stress is the stress acting from the soil on the wheel in the radial direction. It is assumed that the stress is uniform in the lateral direction, thus the normal stress only depends on the angular position on the wheel, \( \theta \), as shown in figure 2.

The Bekker-Reece equation (eq 1) is used to calculate the stress at a given angle \( \theta \). But in eq 1 the depth, \( z \), is used so first the depth has to be expressed in terms of the angle \( \theta \). Using trigonometry the following equation is derived:

\[ z = r(\cos \theta - \cos \theta_f), \]  

(5)

where \( r \) is the radius of the wheel and \( \theta_f \) is the entry angle, see figure 3 below.
Note that the entry angle, $\theta_f$, and the exit angle, $\theta_r$, are not equal in magnitude. This is because the wheel compacts and move the soil to the side after it has passed. The fact that the heights are different means that the sinkage depth, $z$, is dependant on which height is presumed. This is accounted for by estimating the normal stress as a piecewise function \cite{2}:

$$\sigma(\theta) = (ck'_s + b\gamma_k k'_\phi)^{\frac{n}{2}} \left\{ \begin{array}{ll} (\cos \theta - \cos \theta_f)^n & (\theta_m < \theta < \theta_f), \\
(\cos \theta_f - \frac{\theta - \theta_r}{\theta_f - \theta_m} - \phi \theta_f)^n & (\theta_r < \theta < \theta_m), \end{array} \right.$$
θ_m is the angle at which the stress is maximized. The function above is based on equation 1, see [2] for derivation. θ_m is modelled as a linear relation of the slip and entry angle [1].

\[
θ_m = (c_0 + c_1 |s|)θ_r,
\]

where \( s \) is the slip of the wheel, \( c_0 \) and \( c_1 \) are constants determined empirically to be in the range \([0.2, 0.4]\) and \([0.4, 0.5]\) respectively [2].

### 2.1.3 Shear stress

Shear stress is the component of the total stress applied in a direction perpendicular to the normal direction of the area it is applied to. There are two different types of shear stress, tangential and lateral. Tangential shear stress is applied in the tangential direction of the wheel circumference. Both shear stresses are calculated with equations 2 and 3. In order to do so the soil shear displacement, \( j \), has to be estimated. This is done differently for the two types of shear stresses.

To calculate the tangential shear stress it is assumed that the velocity of the soil parcels at the interface between the wheel and the soil is the same as the slip velocity. The slip velocity, \( v_j \), is the tangential component of the total velocity of a point on the wheel which can be expressed as [1]:

\[
v_j(θ) = rω - v_x cos θ = rω(1 - (1 - s) cos θ),
\]

where \( ω \) is the angular velocity. To get the soil displacement, \( j \), the slip velocity is integrated from the point in time when the wheel first touched the parcel, \( t_1 \) to the exit time, \( t_2 \).

\[
j(θ) = \int_{t_1}^{t_2} v_j(θ) dt = \int_θ^{θ_f} r(1 - (1 - s) cos θ)dθ = r(θ_f - θ - (1 - s)(sin θ_f - sin θ))
\]

Combining equation 2, 3 and 8 gives the tangential shear stress, \( τ_x \):

\[
τ_x(θ) = (c + σ(θ) tan φ)(1 - e^{-r(θ_f - θ - (1 - s)(sin θ_f - sin θ)))/k}) \tag{9}
\]

The soil deformation in the lateral direction is estimated using the slip angle, \( β \). The slip velocity in the lateral direction is equal to the lateral velocity component, \( v sin β \). Applying the same method as for the lateral shear stress gives:

\[
j_y(θ) = \int_{θ_1}^{θ_2} v_y(θ)dt = \int_θ^{θ_f} v sin β \frac{1}{ω} dθ = \frac{v sin β}{ω}(θ_f - θ) = r(1 - s)(θ_f - θ) tan(β)
\]

Combining equation 2, 3 and 10 gives the lateral shear stress, \( τ_y \):

\[
τ_y(θ) = (c + σ(θ) tan φ)(1 - e^{-r(1 - s)(θ_f - θ)} tan(β))/k) \tag{11}
\]
2.1.4 Drawbar pull

The drawbar pull, denoted $F_x$, is the net longitudinal force acting on the wheel. It is obtained by integrating the stresses acting in the longitudinal direction.

$$F_x = rb \int_{\theta_e}^{\theta_f} (\tau_x(\theta) \cos \theta - \sigma(\theta) \sin \theta) d\theta$$  \hspace{1cm} (12)

Note that the integral contains two terms. The first term is the integration of the longitudinal shear stress which is governed by friction, this is what drives the vehicle forward during thrust. The other term is the rolling resistance force. This force occurs because of the fact that since the wheel is sunk into the terrain, part of the normal stress acting on the wheel is directed in the negative longitudinal direction. This is why a vehicle is slowed down when sunk into soil.

2.1.5 Lateral force

The lateral force, denoted $F_y$, is the net force acting sideways on the wheel. The force is nonzero only when the wheel has a nonzero slip angle, $\beta$. In other words when the travelling direction is different from where the wheel is directed, for example during turning. The force has two different sources, bulldozing effect and frictional effect beneath the wheel. To get the frictional force, denoted $F_u$ the lateral shear stress is integrated.

$$F_u = \int_{\theta_e}^{\theta_f} \tau_y(\theta) d\theta$$  \hspace{1cm} (13)

2.1.6 Vertical force

The vertical force, denoted $F_z$, is obtained by integrating the stresses acting in the vertical direction.

$$F_z = rb \int_{\theta_e}^{\theta_f} (\tau_r(\theta) \sin \theta + \sigma(\theta) \cos \theta) d\theta$$  \hspace{1cm} (14)

2.1.7 Lateral torque

The lateral torque, denoted $T_y$ is the torque applied around the lateral axis, the axis the wheel is spinning around. The Lateral torque from the soil is resisting the motor torque. It is obtained by integrating the tangential shear stress $\tau_x(\theta)$:

$$T_y = r^2 b \int_{\theta_e}^{\theta_f} \tau_x(\theta) d\theta$$  \hspace{1cm} (15)

2.2 Computer graphics

Only a couple of years ago the graphics processing unit, henceforth GPU, was something that you gave data and it displayed pretty images. But since the GPU is made for making millions of simple calculations in parallel it can be used to great effect for more general purposes, like physical calculations. This is something that has been utilized more and more the last few years and today it is normal to use in simulations, computer and console games.
2.2.1 Graphics pipeline

The graphics pipeline is the algorithm for creating 2D representations on the screen from a 3D scene. Originally the pipeline was the GPUs sole purpose and it could not be used for anything else. The reason for having multiple stages and not just transform the 3D world to the 2D screenspace is to be able to apply different operations to a specific stage. In the simple explanation given below, each stage is seen as a space. The conversion from one stage to the next is preformed using linear algebra matrix computations.

3D model - The model space is the space for a specific 3D model. Usually the origin is in the center of the model.

World - The world space brings together multiple models which are then scaled, rotated and translated.

Camera - Transform the world coordinate system to the camera coordinate system, by translating and rotating so that the camera is in the origin.

Viewport/screen - The viewport is the transformation to the output 2D viewport. Normally the viewport and the screen is the same for a single player game. An example where the viewport- and screen space is different is if a game is played in splitscreen mode.

2.2.2 Shaders

Shaders are programmable applications run in the graphics pipeline where the shaders are run in the pipeline depends on what kind of shader it is. The two most common shaders are vertex and fragment shaders. A vertex shader is run once for every vertex in the mesh of a 3D model. It can for example manipulate the position, color and lighting of a specific vertex.

A fragment shader is run once for every fragment, which is a technical term for pixel. It is run after the rasterization, which is the process of creating pixels from a 3D model. Therefore, a fragment shader cannot deform a surface but it can apply shading depending on the normal vectors of the fragments. Vertex and fragment shaders are run in different orders in the pipeline, but they are commonly paired together. Therefore, it is usual to refer a vertex + fragment program as a single shader even though this is technically two separate shaders.

Another kind of shader is the surface shader, which is specific for Unity. A surface shader is a high level type of shader because it is actually not compiled but rather translated to a resulting vertex + fragment shader. It is a convenient way to apply standard lighting models to surfaces that are normally done directly in a vertex + fragment shader.

A compute shader is a more general type of shader that does not actually belong in the pipeline. A compute shader is instead executed from a command on the CPU and it can output data back to the primary memory or to the memory on
Figure 4: To the left is a ball where lighting is done in the vertex shader, to the right is the same object lit with a fragment shader. Image source: http://dshankar.svbtle.com/lighting-in-unity-5

the GPU. A more in depth explanation of a compute shader will be given in the next section.

2.2.3 General-purpose computing on graphics processing units

The GPU is designed to make millions of simple calculations on big chunks of data quickly and display the data to the screen without saving anything. This is achieved by basically making the same operations on every vertex or fragment which enables parallelization in a way that cannot be achieved by the CPU. This means that the GPU cannot handle conditional branching efficiently. Conditional branching is used to execute different lines of code depending on result of previous executions, for example a if-statement. Furthermore, a shader cannot access data on the primary memory directly, however limited amounts of data can be sent to the shader from a CPU executed script. Previously the GPU could not send data back to the primary memory, but with todays technology it is possible.

A compute shader is a program that is executed on the GPU which enables large amounts of data to be sent from the CPU. Furthermore the computed data can be sent back to the primary memory and used by the CPU. This opens up possibilities for efficient calculation of large amounts of data. There are mainly two drawbacks of making computations on the GPU, first of all sending data back and forth between the GPU and CPU memory is slow, secondly the GPU has to make the same operations for every data element, technically the GPU can make use of branching but it should be avoided since the GPU is not designed for that. In fact, if there is branching in a compute shader the GPU will probably compute both branches and discard the one not used. Doing just one if-statement effectively doubles the computational cost in this case.

2.2.4 Textures

A texture can be described as a image in pixels, one specific coordinate, called texcoord, returns a specific color. A texture is normally applied over a surface
in a game, giving the surface its color in every point. The Unity terrain can have several textures, for example one texture for a grassy area, another for a rocky area and a third for a sandy area. A texture is normally represented as a 2D array where each element is the color for a specific texcoord. A color is described by four values, these values gives the amount of red, green, blue and transparency in that color.

Figure 5: The sandbox level with a terrain that has no texture.

Figure 6: The sandbox level with a terrain that has a single texture.

A render texture is a special type of texture that can be written to and read from in runtime. Render textures does not necessarily need to be used to visualize something, it can also be used to store data on the GPU memory. The advantage of storing data on the GPU memory instead of the primary memory is that computations preformed on the GPU can access this data directly.
2.3 Numerical integration and algorithms

2.3.1 Trapzoidal method

The trapzoidal method is an algorithm for numerical integration by approximating segments of a function \( f(x) \) as linear. The area under a linear segment from \( x = a \) to \( x = b \) is a trapzoid and the expression for this area is:

\[
A = \frac{(b - a)(f(a) + f(b))}{2}
\]  

(16)

The trapzoidal method divides the function into several of these trapzoids and adds the areas beneath each. The algorithm is described below:

\[
h \leftarrow \frac{b-a}{\text{number of segments}}
\]

result \( \leftarrow \frac{h}{2}(f(a) + f(b))\)

for \( i = 1 \) to number of steps do

result \( \leftarrow \text{result} + hf(a + i \times h)\)

2.3.2 Perlin noise

Perlin noise can be used to add seemingly random and continuous noise to a surface. The algorithm for creating 2D noise is described briefly below.

The base of the perlin noise is a two-dimensional grid where each node has a random vector assigned. For a specific point it has to be determined which grid the point is in and a distance vector from the point to the grid corners is computed. The dot product of between the distance vector and the random vector is computed for all four nodes in the grid. Finally a interpolation between the four dot products is computed.

2.3.3 Grid Interpolation

Grid interpolation is a method of constructing new data points using a set of given data points in two dimensions. The basic idea is to assume a function and adapt a set of parameters in order to fit the original data points. A basic overview of two different types of grid interpolation will be given below.

Bilinear interpolation is based on linear interpolation. The surface is divided into a grid where the corners in the grid are data points. A specific point on the surface lies in a grid and the value of the point is determined using the data points in the corners of the square.

Linear interpolation in the \( x \)-direction:

\[
f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}),
\]  

(17)

\[
f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}),
\]  

(18)

where \( Q_{XX} \) is the data point in \((x_X, y_X)\) and \( f(x, y) \) is the function resulting from the interpolation. The next and last step is to perform interpolation in
the y-direction:

\[
f(x, y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)
\]  \hspace{1cm} (19)
3 The tech demo

This section will review how the actual application uses the theory to create a working product. The tech demo is developed in Unity and makes use of several of its utilities. It has two objects of interest, the terrain and the vehicle. A explanation of how these objects are built and how they work will be given, then a summary of how the physical computations are made using the theory.

3.1 The vehicle

![Figure 7: The ATV in the tech demo.](image)

The vehicle is a four-wheeled all-terrain vehicle, henceforth ATV. The ATV is a Unity-object, which is a class in Unity that works like a blueprint for how an instance of this object will be built, exactly like how classes in object-oriented programming languages are blueprints for objects of a specific type. The ATV is a very simple vehicle built of four different components, a collider, a rigidbody, four wheel-colliders and a script.

A collider is a component that handles collisions with other colliders and the terrain. Most colliders take up a specific volume, like a sphere or a cube, and if there is any volumetric overlap with any other collider a collision is detected.

A rigidbody is a component that handles the rigidbody physics of an object. This includes how the object reacts after a collision is detected. The rigidbody calculates forces and torques from collisions which is then used to give an object its dynamic behaviour. Forces can also be added explicitly to rigidbodies through scripts; this is how the forces returned from the model described in section 2.1 are used.
A wheel collider is a special type of collider that are meant to be used for a vehicle’s wheels. It keeps track of properties such as slip, slip angle and rotational speed. Normally a collider does not handle any physics because but the wheel collider is different since it use the data it calculates to add forces and torques to the rigidbody it is attached to, for example the traction force. The wheel collider needs certain properties which are given by the user in a interface, see figure 8.

A script is a component that is written in either C# or JavaScript, it is a sequence of code that governs a specific behaviour of the object. For example if the vehicle should move when a button on the keyboard or controller is pressed.

![Figure 8: Wheel collider properties window in Unity.](image)

### 3.2 The terrain

The terrain object in Unity is a asset which provides several useful tools for creating vast landscapes and handles collision detection efficiently. Its base is a heightmap with a graphical mesh on it. The user has access to the heightmap by using built-in functions that returns an array of heightdata, but not to the graphical mesh. See figure 6 for a image of the terrain used in thesis thesis.

### 3.3 Physics calculation

The sinkage of a wheel into the terrain is calculated by emposing the condition that the vertical force from the wheel down on the terrain is equal to the force
from the terrain on the wheel. The integral in equation 14 cannot be solved explicitly so it has to either be solved numerically or approximated to something that can be integrated analytically. Here the first method is used. The algorithm for solving the sinkage works like this:

1. Set the sinkage and normalForce to 0.
2. while normalForce < vertForceFromWheel do the following:
   - increment sinkage by a small value.
   - calculate new entry and exit angles.
   - calculate new normalForce.

Once the entry and exit angles are computed the shear and normal stresses can be calculated. All of the vehicle forces and torques are computed by integrating normal and/or shear stresses. All of these integrations has to be done numerically with the trapzoidal method. With these stresses the wheel forces can be calculated using the theory in section 2.1. These wheel forces and torques are then applied to a point on the vehicle rigidbody in Unity. The calculation of the sinkage is made once every timestep for each wheel, the sinkage considered is the maximum sinkage into the terrain of the wheel at a certain time.

3.4 Graphical terrain deformations

There is a distinction between the graphical and physical deformations of the terrain in the tech demo. The graphical deformations are what can be seen. When the terrain get deformed it is displayed on the screen, see figure 9. The other type of deformation is the physical deformations. This is the deformation of the actual heightmap which handles collision detection with the vehicle, see figure 9.

The reason for this distinction is that the graphical deformation data can be stored on the GPU. This is good because these deformations are calculated on the GPU and if the data can be accessed directly by the GPU there is no need to send this data from the primary memory. This way extremely detailed deformations can be computed for low computational cost every frame. However, there are a few drawbacks to this method. First of all, the GPU memory is about one forth of the primary memory on most machines and the GPU memory is needed for other purposes. Secondly, the graphical mesh on the terrain object in Unity cannot be manipulated by a shader (technically it actually can but the problem is that it is impossible to control where deformations occur from a vertex shader because the vertices are scattered ununiformly across the mesh).

3.4.1 The tiles

These problems are solved by using a small plane object, henceforth called a "tile". A tile is basically just a small mesh with a texture on it, the mesh can easily be manipulated by a vertex shader to create deformations. The point of the tile is to replace a small part of the graphical mesh of the terrain so that
Figure 9: Above is a picture of the physical deformations without the graphical layer, below is the graphical deformations of the same terrain patch.

graphical deformations can be used.
Figure 10: Tiles are placed where the ATV has travelled to visualize deformations. The blue area is one single tile.

Figure 11: If the shading is not equal on the terrain and the tiles a edge can be seen.

This method solves the problem of graphical deformations on the Unity terrain, but creates another problem. This is that since the tiles replace a small part of the terrain, it has to look exactly the same as the part of the terrain that was replaced. For one, the shading has to be the same otherwise the player can see a edge between the tile and the terrain, see figure 11. This problem can be solved by applying the surface shader used by Unity on the terrain on the tiles. Another problem is that the texturing has to be the same on the tile and the terrain. Using the same texture on the tile and the terrain is obvious, but the texcoords has to be manipulated on the tile in order to fit into the rest of the terrain. The texcoords have to be offset and scaled by the position and size of the tile relative to the whole texture.

The purpose of the tiles is to visualize the terrain deformations. Hence the tiles has to be placed where the terrain deformations are. These deformations are located in the tracks of the vehicle, so the tiles has to be placed to cover the areas where the tracks are. If the vehicle is driving around in the terrain for a long enough time, eventually there will be too many tiles for the GPU to
handle, the machine will simply run out of GPU memory and crash. In order to solve this issue there is a maximum number of tiles that can be used. When the vehicle has travelled long enough the tile that the vehicle last visited will be reset and moved to the position in front of the vehicle. This is done by using a buffer of a fixed number of tiles and implementing a first in first out queue system to keep track of which tile was visited last.

3.4.2 The compute shader

The compute shader has two purposes, first of all it has to calculate the deformations and write this data to the render texture. Secondly, it has to read heightdata from the terrain heightmap and write this to the render texture when a tile is moved to a new position. This is done in two different functions of the compute shader, called kernels.

The first kernel that computes deformations takes the sinkage into the terrain, position of the wheel, direction of the wheel and the original height of the tile as input. It then calculates new heights for every fragment of the area around the wheel. These heights are calculated in two steps. First, the deformation around the wheel is approximated as a the 2nd derivative gaussian that can be described with the following equation:

\[ H(x, y) = \left( \frac{x^2}{\sigma^2} - 1 \right) e^{-\frac{x^2 + y^2}{2\sigma^2}} \frac{1}{2\pi\sigma^4}, \]  

where \( \sigma \) is the width of the gaussian. The peaks are meant to be at the sides of the wheel, when a wheel is travelling in soft terrain the soil is pushed up at the sides of the wheel. The function is scaled with the sinkage in order to get the correct depth of the tracks. The deformation governed by this equation is only used for visual effects and does not affect the physical heightfield that is used for collision detection. The space used in equation 20 is the local space of the deformation, which means \( y \) is the forward direction of the wheel and \( x \) is the sideways direction. So a rotational variable change has to be made depending on

\[ H(x, y) = \left( \frac{x^2}{\sigma^2} - 1 \right) e^{-\frac{x^2 + y^2}{2\sigma^2}} \frac{1}{2\pi\sigma^4}, \]  

\[ 18 \]
the direction of the wheel in order to use the world coordinates. The second step of the height computation is to apply bump mapping in the form of perlin noise to make the deformed soil look more realistic. The noise is multiplied by the difference in original height (before any deformations) and the current height. This way the noise is more prevalent where the deformations altered the terrain.

The second kernel that applies heightdata to the render texture takes a array of floats as indata. These floats are extracted from the heightfield of the terrain object. The resolution of the heightmap is generally not the same as the resolution of the render texture. Imagine that the tile is 10 x 10 meters and the resolution is 10 data points per meter which gives 100 x 100 data points from the heightfield for one tile, but the resolution of the render texture is about 1000 x 1000 pixels. This means the heights of the render texture pixels has to be computed using interpolation between the data points from the heightfield. Bilinear interpolation is used for this purpose.

3.5 Physical terrain deformations

A physical deformation is a deformation of the terrain collider that the vehicle interacts with. This can be done using a builtin Unity function called SetHeights. The problem with this function is that it is too slow to be used in realtime. The physical deformations do not need nearly as high resolution as the graphical deformations, which is partly why they are separated. The manipulation of the heightmap is calculated in the same compute shader as the graphical deformations, but instead of saving the data on the GPU memory the data is sent back to the primary memory.
4 Result

This section will present some results showing the vehicle model performance, computational efficiency and visual effects.

Below is a comparison of the tech demo deformations and real ATV tracks in a sandy terrain.

![Figure 13: Track deformations from the tech demo.](image1)

![Figure 14: Real track deformations from a sandy terrain. Image source: http://rugspot.deviantart.com/art/Sand-Tracks-296629519](image2)

Below is a snapshot of the profiler when the graphical deformations are calculated on the compute shader and when they are computed using the builtin function SetHeightDelayLOD. To the left of the red line in figure 15 the deformations are computed using the builtin function SetHeightDelayLOD, to the right the deformations are computed with the compute shader. Furthermore, figure 16 shows the profiler with the deformations again calculated using the
compute shader but on a bigger terrain.

**Figure 15:** Profiler showing the computation time for each frame.

**Figure 16:** Profiler showing the performance of the program run on a bigger terrain.

The figure below is showing the distribution of time of the different parts of the program when it is run on a machine with approximately the same performance as the next generation of consoles.

**Figure 17:** The distribution of time among the program's functions and operations.

Below is a figure showing the slip of one wheel during full acceleration from standstill to about 20 m/s
5 Extensions and improvements

5.1 Multipass effect

Multi-pass in this case means multiple loadings of a specific patch on the terrain. When a wheel is passing a patch it will change the properties of the soil, the most obvious change is that of the density, $\gamma$, which will increase as the soil is compacted. Other parameters that might change are the cohesion, $c$, and the soil deformation modulus, $k$. The changes to these properties can be expressed by the following equations according to [4]:

$$X_n = X_0(1 + (1 - e^{-s_0k_1})k_2 + k_3n_p),$$  \hspace{1cm} (21)

where $X$ is any of the properties $\gamma$, $c$ or $k$, $X_{np}$ is this property after $n_p$ passes, $s_0$ is the slip of the last pass and $k_1$, $k_2$, $k_3$ are experimentally determined constants.

5.2 Custom wheel collider and vehicle physics

The indata for the physics model in this thesis is the slip, slip angle and vertical load from the wheel on the terrain. A wheel collider is an object which collides with the terrain and returns information about the collision, for example the force applied in the contact. The default wheel collider in Unity is used for this purpose. The drawback of using this default wheel collider is the lack of control over how the slip and wheel forces are calculated. The Unity wheel collider calculates its own drawbar pull and side force which cannot be disabled which means the forces and torques calculated with the model cannot be used as long as the builtin wheel colliders are used. In order to properly implement
the physical theory given in this thesis a custom wheel collider has to be built and used.

A custom wheel collider would be built to simply return indata needed for the model. One possible way to estimate the vertical force from the wheel on the terrain would be to divide the force in two terms, the static load and the shock force. The static load can be calculated using the spring connecting the wheel and the vehicle body and the shock force could be calculated using the velocity of the wheel. If the wheel in one time step has a high downwards velocity and in the next timestep has no downwards velocity because it collided with the terrain, a force had to decelerate the wheel. The deceleration can be calculated using Newtons second law of motion.

The slip and slip angle has to be calculated by implementing proper vehicle physics. The slip is calculated trough its definition [ref], which requires the rotational speed of the wheel, \( \omega \). \( \omega \) depends on the engine torque, gear ratio and other engine parameters.

5.3 Custom heightfield

At the moment the tech demo make use of a Unity built in function to change the heights of the terrain. This is a problem because the function that does this is too slow for realtime applications. A better approach would be to build a custom heightfield. This heightfield would only interact with the custom wheel colliders described above, that way a geometric object would not have to be deformed, only the values of the heightfield.

The heightfield would just be a 2D array of heightdata, much like the render textures are heightdata for the graphical tiles. This way if the position of the wheel is known, the height beneath the wheel can easily be extracted from the heightfield.

5.4 Deformable areas

The tech demo is built with a single terrain object and constant soil parameters over the whole terrain. This means there is no possibility to have several different types of terrain in the same scene. Some areas in the game might not even be deformable at all, in that case there is no need for the tiles to replace the terrain in these areas.
6 Discussion

Figure 18 in the result section shows the slip versus time for the vehicle in acceleration. The reason why I showed this plot is to illustrate that the vehicle dynamics in this tech demo is not very stable. In this model the slip is supposed to depend on the force and thus indirectly the acceleration of the vehicle, so the slip ratio is clearly wrong. The reason for this may be because the terrain collider is somehow used by unity to calculate slip and slip angle and since this is being deformed beneath the wheel the calculations become unstable. Also the wheel colliders calculates their own torques and forces which means the forces and torques calculated using the theory cannot be utilized, at the moment the vehicle forces and torques (like drawbar pull) are all calculated with the builtin wheel colliders. Furthermore, the wheel colliders only returns a single contact point, if the wheel has multiple contact points only the point which had contact first will be used, the deformations will thus be calculated as if the wheel only had contact with that point. In order to have full control over all the forces and torques on the wheels and how that affect the vehicle one must create custom wheel colliders, as explained in section 5.2. This requires implementation of a drivetrain model.

One of the goals of the thesis was from the beginning to create deformable terrain and a vehicle that interacts with the terrain in a realistic manner. Unfortunately this could not be achieved because the focus of the thesis became somewhat different during the process of developing the tech demo. The main focus has been to implement a method that will work in a real computer game environment and not just in a small sandbox level. Imagine that I would have developed a method that used physics to deform the terrain and how the vehicle interacts with it and I had used a small terrain and the builtin function SetHeightsDelayLOD. The physics may have been relatively realistic but the method in which it was developed would not be possible to implement in a real game because it is too slow for big terrains. In that case all the work done by me would have mostly wasted. Instead I choose to, in agreement with my supervisor, focus on developing a method for computing deformations efficiently that can work in a big environment.

The new focus is the way the GPU is used to calculate deformations, how the data is stored on the GPU and how the CPU and GPU communicate. The idea was to create a method where the size of the terrain does not matter. This is why there is a constant buffer of tiles which are used to deform the terrain. The bigger the terrain is the bigger the difference in computational efficiency between my GPGPU method and the using the builtin function.

Figure 15 is showing the profiler of a simulation. The first part is run without the graphical tiles and only using the function SetHeightsDelayLOD to deform the terrain both graphically and physically. The second part is run with the tiles and without using the function SetHeightsDelayLOD. As can be seen the resulting computational efficiency is notably better even in a small terrain like the sandbox level.
7 Conclusions

"The primary goal of this project is to develop a tech demo of a game where a vehicle is moving in a small landscape with dynamically deformable terrain."
As shown in figure 13 the vehicle is indeed moving in a small landscape and deforming the terrain.

"The vehicle should interact with the terrain in a realistic manner."
A plot showing the slip behaviour is presented in figure 18. This plot shows that the vehicle interaction with the terrain is not realistic because the slip should not change sign during acceleration. Further expansions that would solve this issue is presented in section 2.1.

"The game has framerate requirement of 60 frames per second which is approximately 16 ms per frame. The other parts of the game is approximated to take 10-12 ms in the worst case scenario. This means that the tech demo should be implemented in such a way so that the underlying theory allows for realtime execution below 5 ms per frame. However, this requirement does not apply to the implementation since this will be subject to optimization by professionals at this area. This allows for the focus of the thesis to be directed towards the development of the algorithm."

The tech demo is developed in such a way that bigger terrain does not increase the computation time of the calculations, this is shown when comparing figures 15 and 16. Also when the program is run on a machine comparable to the next generation of consoles, the time consumption for a frame is below 5 ms.
A Appendix - Code

A.1 Script - deformTerrain

using UnityEngine;
using System.Collections;

// Terrain constants and parameters class.
struct TerrainParams{
    public float cohMod { get; set; }
    public float friMod { get; set; }
    public float defExp { get; set; }
    public float cohStress { get; set; }
    public float density { get; set; }
    public float friAngle { get; set; }
    public float sheerDefModulus { get; set; }
    public float kappa { get; set; }
    public float diggingIndex { get; set; }
    public Terrain west { get; set; }
    public Terrain east { get; set; }
    public Terrain north { get; set; }
    public Terrain south { get; set; }
    public float terrainScale { get; set; }
    public int hmResolution { get; set; }
}

class DeformPattern{
    public DeformPattern()
    {
        Pos = new IntVec2();
        Size = new IntVec2();
        CirclePos = new IntVec2();
    }
    // Calculate how the heightsbox for deformation should be shaped and placed depending on where the wheel collision is.
    public IntVec2 Pos { get; set; }
    public IntVec2 Size { get; set; }
    public IntVec2 CirclePos { get; set; }
}

class deformTerrain : MonoBehaviour {
    // Object references
    private WheelHit hit;
    private GameObject terrainGO;
    IntVec2 planePos;
    IntVec2 oldPlanePos;
    private Terrain terrain;
TerrainParams tp = new TerrainParams();

public Rigidbody rb { get; protected set; }
public Rigidbody[] wRB { get; set; }
public float pressure { get; set; }
public float maxShearStress { get; set; }

// Position and resolution variables.
Vector2 terrainNormalizeCoeff; // heightmap width and height divided by size of the terrain
float[,] heights;
int hmWidth, hmHeight; // heightmap width and height respectively
int renderTextureRes = 1020;
int tileSize = 10;

// Wheel and vehicle variables and parameters.
float staticSinkage;
Vector3 velocity;
float slipAngle;
float slip;
float entryAngle;
float exitAngle;
float thetaM;
float a0;
float a1;
float verticalForce;
float TotalSideForce;
public float wheelWidth;
public float appliedForce;
float contactArea;
float tractiveForce;
[SerializeField] int deformRadius = 2;
int defSquareSize = 200;
Density terrainProperties;

Vector2 step;
IntVec2 deformPos;

public void InitializeTerrain (DeformInfo[] deformInfo)
{
    GetComponent<tileController>().InitializeTileBuffer(30);

    rb = GetComponent<Rigidbody>();
wRB = new Rigidbody[deformInfo.Length];
wRB = GetComponentsInChildren<Rigidbody>();
a0 = 0.4f;
a1 = 0.2f;
slip = 0;
staticSinkage = 0.0001f;

step = new Vector2();
deformPos = new IntVec2();

oldPlanePos = new IntVec2(-1, -1);
planePos = new IntVec2();

public void updateTerrain(DeformInfo deformInfo)
{

hmWidth = deformInfo.curTerrain.terrainData.
    heightmapWidth;
hmHeight = deformInfo.curTerrain.terrainData.
    heightmapHeight;

terrainNormalizeCoeff.x = hmWidth / deformInfo.
    curTerrain.terrainData.size.x;
terrainNormalizeCoeff.y = hmHeight / deformInfo.
    curTerrain.terrainData.size.z;

// Get information from the terrain.
terrainProperties = deformInfo.curTerrain.
    GetComponentInParent<Density>();

tp.density = terrainProperties.GetComponent<Density
    >().density;
tp.diggingIndex = terrainProperties.GetComponent<
    Density>() .diggingIndex;
tp.cohMod = terrainProperties.GetComponent<Density
    >().cohMod;
tp.friMod = terrainProperties.GetComponent<Density
    >().friMod;
tp.defExp = terrainProperties.GetComponent<Density
    >().defExp;
tp.kappa = terrainProperties.GetComponent<Density
    >().kappa;
tp.friAngle = terrainProperties.GetComponent<Density
    >().friAngle;
tp.sheerDefModulus = terrainProperties.GetComponent<
    Density>() .sheerDefModulus;
tp.cohStress = terrainProperties.GetComponent<Density
    >().cohStress;
tp.terrainScale = deformInfo.currTerrain.GetComponentInParent<Terrain>().terrainData.heightmapScale.y;
tp.hmResolution = deformInfo.currTerrain.GetComponentInParent<Terrain>().terrainData.heightmapResolution;
}

public void applyDeformation(DeformInfo[] deformInfo) {
    // Temp variables
    Vector2 coord;
    int numWheels = deformInfo.Length;
    int deformDiam = deformRadius * 2;
    DeformPattern dp = new DeformPattern();
    for (int wheelNo = 0; wheelNo < numWheels; wheelNo++) {
        // Shorthand variable
        DeformInfo di = deformInfo[wheelNo];

        // Debug.Log("wheelNo: " + wheelNo);
        // Debug.Log("onGround: " + di.onGround);
        if (di.onGround) {
            updateTerrain(di);

            // Set positional variables of the wheel-terrain collision.
            // Construct a vector containing the x- and z-coordinate difference
            // Scale with the size of the terrain
            coord = Vector2.Scale((di.Pos - new Vector2(di.currTerrain.gameObject.transform.position.x, di.currTerrain.gameObject.transform.position.z)), terrainNormalizeCoeff);

            // coord = (di.Pos); // - new Vector2(terrain.gameObject.transform.position.x, terrain.gameObject.transform.position.z));

            // Typecast the tempCoord to int
            di.currPos = new IntVec2((int)coord.x, (int)coord.y);

            planePos.x = (int)di.Pos.x / tileSize;
            planePos.y = (int)di.Pos.y / tileSize;
        }
    }
}
Vector2 meshPos = new Vector2(planePos.x, planePos.y);
Vector2 renderTexturePos = new Vector2((di.Pos.x - meshPos.x * tileSize) * renderTextureRes/tileSize, (di.Pos.y - meshPos.y * tileSize) * renderTextureRes/tileSize);

// Check if the vehicle has moved
if ((Mathf.Abs(di.currPos.x - di.prevPos[0].x) > 0 ||
Mathf.Abs(di.currPos.y - di.prevPos[0].y) > 0)) {

// Shorthand variable
float r = di.radius;
// Declare local variables
float speed; // Speed of the vehicle.
float sinkage; // How much the vehicle sinks

// Temporary variables for storing different coordinate systems
Vector3 wheelForwardDir; // Vector pointing in the wheels forward direction.
Vector3 sidewaysDir; // Vector perpendicular to the vector above.
Vector3 steeringPlaneNormal; // Normal vector to the plane in which the wheel is steering.
Vector3 velSteeringPlane; // Vehicle velocity projected onto above mentioned plane.

// Get density from the terrain, this value may be updated depending on deformation
tp.density = terrainProperties.GetComponent<Density>().density;

// Get information from the vehicle and the wheel
// Get the slip value from the WheelHit
slip = -di.fwdSlip;
// Get the velocity in world space
velocity = rb.velocity;

// Get directions in world space
wheelForwardDir = di.fwdDir; // Forward (+z) of the wheel

// Calculate the sideways direction (the wi.wh. sidewaysDir does not work properly) and normalize
sidewaysDir = Vector3.Cross(Vector3.Cross(velocity, wheelForwardDir), wheelForwardDir).normalized;

// Calculate the speed
speed = Vector3.Magnitude(velocity);
// Calculate normal vector to the plane in which
// the wheel is steering.
steeringPlaneNormal = Vector3.Cross(
    wheelForwardDir, rb.transform.right).normalized;
// Vehicle velocity projected onto
steeringPlaneNormal

// Calculate the slip angle
slipAngle = Mathf.Acos(Vector3.Dot(wheelForwardDir, velSteeringPlane));
// wheelSpeed = r * wi.rpm * 2 * 3.1415f/60; // Original line
// wheelSpeed = r * di.rpm * 0.10472f;

// Calculate the contact area
// We approximate the contact angle to 0.1 rad ~
// 6 deg. This gives 0.1/(2*pi) ~ 0.016
contactArea = wheelWidth*r;
// Calculate static sinkage and corresponding
// entry/exit angles etc.
pressure = di.force; // Static pressure
// Set initial sinkage to some constant low
// value.
sinkage = staticSinkage;

// Calculate the entry and exit angle
entryAngle = Mathf.Acos((1-tp.terrainScale*sinkage/r));
exitAngle = -Mathf.Acos((1-tp.kappa*tp.terrainScale*sinkage/r));
// Calculate the angle beneath the wheel where
// the normal stress is the largest
thetaM = (a0 + a1 * slip)*entryAngle;
// Calculate the vertical force
verticalForce = wheelWidth * r * (entryAngle - exitAngle) * NormalStress(thetaM, r) / 2;

// Calculate dynamic sinkage and final entry/
// exit angles.
while (verticalForce < 0.99f*di.force)
    sinkage += 0.00005f;
entryAngle = Mathf.Acos(1 - tp.terrainScale * sinkage / r);
exitAngle = -Mathf.Acos(1 - tp.kappa * tp.terrainScale * sinkage / r);
thetaM = (a0 + a1 * slip) * entryAngle;

//vertical Force = wheelWidth * r * (entryAngle - exitAngle) * NormalStress(thetaM, r) / 2;
verticalForce = wheelWidth * r * IntegrateTrapezoidal(exitAngle, entryAngle, NormalStress, 50, r);

// Calculate how the deformation pattern should look and how heightsbox for deformation should be shaped and placed depending on where the wheel collision is.
dp.Pos = di.currPos - new IntVec2(deformRadius);
dp.Size = new IntVec2(deformDiam);
dp.CirclePos = new IntVec2(deformRadius);

// A bunch of if-statement to check if the collision is close to an edge of the terrain
if (di.currPos.y >= hmHeight - deformRadius)
{
dp.Size.y = deformRadius + hmHeight - di.currPos.y;
}
if (di.currPos.x >= hmWidth - deformRadius)
{
dp.Size.x = deformRadius + hmWidth - di.currPos.x;
}
if (di.currPos.y <= deformRadius)
{
dp.Pos.y = 0;
dp.Size.y = di.currPos.y + deformRadius;
dp.CirclePos.y = di.currPos.y;
}
if (di.currPos.x <= deformRadius)
{
dp.Pos.x = 0;
dp.Size.x = di.currPos.x + deformRadius;
dp.CirclePos.x = di.currPos.x;
}

// Get the current heights from the terrain
heights = di.currTerrain.terrainData.GetHeights(dp.Pos.x, dp.Pos.y, dp.Size.x, dp.Size.y);

// Set new heights.
for (int i = 0; i < heights.GetLength(0); i++)
{
    for (int j = 0; j < heights.GetLength(1); j++)
    {

// The shape of the deformation around the collision point is a circle.
// First condition is to apply the deformation as a circle around the collision point.
if ((Mathf.Pow(i - dp.CirclePos.y + 1, 2) + 
    Mathf.Pow(j - dp.CirclePos.x + 1, 2)) 
    <= Mathf.Pow(deformRadius, 2))
  
  // Set the new height as the old value minus how much we sink and dig down
  heights[i,j] = heights[i,j] - sinkage; // - digging;
}
}
di.currTerrain.terrainData.SetHeightsDelayLOD(
    di.currPos.x - deformRadius, di.currPos.y - deformRadius, heights);

// If the current wheels position is on a new tile, change the tile that is going to get deformed.
if (planePos.x != oldPlanePos.x || planePos.y != oldPlanePos.y){
    GetComponent<tileController>().ChangeCurrentTile(planePos);
}

// If wheel is close to the edge of the tile a tile needs to be put on that side since deformations can occur on neighbouring tiles.
// Check if the wheel is close to the edge to the...
// North
if (((int)(di.Pos.y + (float)defSquareSize/(float)renderTextureRes*5))/tileSize !=
    oldPlanePos.y){
    GetComponent<tileController>().ChangeTile(new IntVec2((int)di.Pos.x/tileSize, (int) (di.Pos.y + (float)defSquareSize/(float)renderTextureRes*5)/tileSize),1);
}

// Northeast
if (((int)(di.Pos.y + (float)defSquareSize/(float)renderTextureRes*5))/tileSize !=
    oldPlanePos.y && (int)(di.Pos.x + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize != oldPlanePos.x) {
  GetComponent<tileController>() . ChangeTile(new
  IntVec2((int) (di.Pos.x + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize , (int) (di.Pos.y + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize ), 2);
}

// East
if ((int) (di.Pos.x + (float)defSquareSize/(float) renderTextureRes*5)/tileSize !=
oldPlanePos.x) {
  GetComponent<tileController>() . ChangeTile(new
  IntVec2((int) (di.Pos.x + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize , (int) di.Pos.y/tileSize ), 3);
}

// Southeast
if ((int) (di.Pos.y - (float)defSquareSize/(float) renderTextureRes*5)/tileSize !=
oldPlanePos.y && (int) (di.Pos.x + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize != oldPlanePos.x) {
  GetComponent<tileController>() . ChangeTile(new
  IntVec2((int) (di.Pos.x + (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize , (int) (di.Pos.y - (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize ), 4);
}

// South
if ((int) (di.Pos.y - (float)defSquareSize/(float) renderTextureRes*5)/tileSize !=
oldPlanePos.y) {
  GetComponent<tileController>() . ChangeTile(new
  IntVec2((int) di.Pos.x/tileSize , (int) (di.Pos.y - (float)defSquareSize/(float)
  renderTextureRes*5)/tileSize ), 5);
}

// Southwest
if ((int) (di.Pos.y - (float)defSquareSize/(float) renderTextureRes*5)/tileSize !=
oldPlanePos.y && (int) (di.Pos.x - (float)
defSquareSize/(float) renderTextureRes*5)/
tileSize != oldPlanePos.x) {

GetComponent<tileController>().ChangeTile(new
IntVec2((int)(di.Pos.x - (float)defSquareSize/(float)renderTextureRes*5)/
tileSize, (int)(di.Pos.y - (float)defSquareSize/(float)renderTextureRes*5)/
tileSize),6);

//West
if (((int)(di.Pos.x - (float)defSquareSize/(float)renderTextureRes*5)/tileSize !=
oldPlanePos.x)) {
GetComponent<tileController>().ChangeTile(new
IntVec2((int)(di.Pos.x - (float)defSquareSize/(float)renderTextureRes*5)/
tileSize, (int)di.Pos.y/tileSize),7);
}

//Northwest
if (((int)(di.Pos.y + (float)defSquareSize/(float)renderTextureRes*5)/tileSize !=
oldPlanePos.y && (int)(di.Pos.x - (float)defSquareSize/(float)renderTextureRes*5)/
tileSize != oldPlanePos.x)) {
GetComponent<tileController>().ChangeTile(new
IntVec2((int)(di.Pos.x - (float)defSquareSize/(float)renderTextureRes*5)/
tileSize, (int)(di.Pos.y + (float)defSquareSize/(float)renderTextureRes*5)/
tileSize),8);
}

//This if statement checks if the distance
between the position wheel this timestep and
the last timestep.
//If the distance is long enough there has to
be deformations applied in between the
deforation this timestep
//and the previous timestep.
if (Mathf.Abs(Vector2.SqrMagnitude(di.Pos - di.
oldPos)) > 0.02f && di.prevOnground) {

//Vector that points from the old wheel
position to the current.
Vector2 step = di.oldPos - di.Pos;
float stepSize = 0.1f;

//Calculate how many additional deformations
has to be done with a fixed stepsize.
float nSteps = Mathf.Sqrt(step.sqrMagnitude) / stepSize;

// Apply all the additional deformations
// TODO, this is very inefficient since
DeformTerrain is called many times and it
is quite expensive since it gets all the
height data for the whole tile every single
time DeformTerrain is called.
// A possible optimazation is to create a
different method that creates all of these
additional deformations in one go or even
better: build this functionality into the
DeformTerrain method.
for (int i = 0; i < nSteps; i++){
    GetComponent<tileController>().DeformTerrain(
        renderTexturePos + (step*i / nSteps)*renderTextureRes/tileSize,
        sinkage * tp.terrainScale, defSquareSize,
        new Vector2(velSteeringPlane.x, velSteeringPlane.z));
}
}
else{
    GetComponent<tileController>().DeformTerrain(
        renderTexturePos, sinkage * tp.
        terrainScale, defSquareSize, new Vector2(velSteeringPlane.x, velSteeringPlane.z));
}

// If the collision is close to the edge of
another terrain object it should deform that
object as well.
/* if (di.currPos.y >= hmHeight - deformRadius){
    deformNeighbourTerrain(tp.west, 1,
        deformRadius - (hmHeight - di.currPos.y),
        dp.Pos.x, dp.Size.x, sinkage);
}*/
if (di.currPos.x >= hmWidth - deformRadius){
    deformNeighbourTerrain(tp.north, 2,
        deformRadius - (hmWidth - di.currPos.x),
        dp.Pos.y, dp.Size.y, sinkage);
}
if (di.currPos.y <= deformRadius){
    deformNeighbourTerrain(tp.east, 3,
        deformRadius - di.currPos.y, dp.Pos.x, dp.
        Size.x, sinkage);
}
if (di.currPos.x <= deformRadius){

deformNeighbourTerrain(tp.south, 4, 
    deformRadius – di.currPos.x, dp.Pos.y, dp.
    Size.y, sinkage);
}
*/

oldPlanePos.x = planePos.x;
oldPlanePos.y = planePos.y;

// Replace the old position with the new for
next timestep
di.prevPos[2] = di.prevPos[1];
di.prevPos[1] = di.prevPos[0];
di.prevPos[0] = di.currPos;

di.oldPos = di.Pos;

di.prevOnground = true;

// Calculate total sideways force on the wheel ( 
  lateral stress beneath the wheel plus
  bulldozing force).
TotalSideForce = IntegrateTrapezoidal(exitAngle, 
  entryAngle, TotalSideStress, 50, r);

// Safeguard against NaN-value for the force
if (float.IsNaN(TotalSideForce))
    TotalSideForce = 0;

// Add the sideforce in the sideways direction
// rb.AddForce(TotalSideForce * sidewaysDir);

if (wheelNo == 0){
    using (System.IO.StreamWriter file = 
            new System.IO.StreamWriter(@"Z:\persistent\ 
                Jens\viktors\deformterrain\slip_data.txt", 
                true))
    {
        file.WriteLine(di.fwdSlip.ToString() + "\n" + velocity.magnitude.ToString();
    }
}
else{
    di.prevOnground = false;
}
private float NormalStress(float theta, float r) {
    // Normal stress from the terrain on the wheel.

    float a0 = 0.4f;
    float a1 = 0.2f;
    float kc = tp.cohMod;
    float kt = tp.friMod;
    float b = wheelWidth;
    float c = tp.cohStress;
    float rho = tp.density;
    float n = tp.defExp;
    float s = slip;
    float thetaM = (a0 + a1 * s) * entryAngle;

    if (thetaM <= theta && theta < entryAngle) {
        return (c * kc + rho * b * kt) * Mathf.Pow(r / b, n) * Mathf.Pow(Mathf.Cos(theta) - Mathf.Cos(entryAngle), n);
    }

    if (exitAngle < theta && theta < thetaM) {
        return (c * kc + rho * b * kt) * Mathf.Pow(r / b, n) * Mathf.Pow(Mathf.Cos(entryAngle - (theta - exitAngle) / (thetaM - exitAngle)) * (entryAngle - thetaM) - Mathf.Cos(entryAngle), n);
    }

    return 0;
}

private float LateralSheerStress(float theta, float r) {
    // Lateral (z axis in object space) shear stress from the terrain on the wheel.

    float c = tp.cohStress;
    float fi = tp.friAngle;
    float ky = tp.shearDefModulus;
    float beta = slipAngle;
    float s = slip;

    return (c + NormalStress(theta, r) * Mathf.Tan(fi)) * (1 - Mathf.Exp(-r * Mathf.Abs(Mathf.Tan(beta)) * (1 - s) * (entryAngle - theta) / ky));
}

private float D1(float Xc, float fi) {
    // Help function for the lateral shear stress.
}
return Mathf.Cos(Xc) / Mathf.Sin(Xc) + Mathf.Tan(Xc + fi);
}

private float D2(float Xc, float fi){
    //Helpfunction for the lateral sheers stress.
    return Mathf.Cos(Xc) / Mathf.Sin(Xc) + Mathf.Pow(Mathf.Cos(Xc) / Mathf.Sin(Xc), 2) / Mathf.Cos(fi) / Mathf.Sin(fi);
}

private float BulldozingForce(float theta, float r){
    //The force that occurs because the wheel displaces the terrain.
    float fi = tp.friAngle;
    float rho = tp.density;
    float c = tp.cohStress;
    float Xc = (1.570795f - fi) * 0.5f; // float Xc = 3.1415f / 4 - fi / 2;
    float h = r * (1 - Mathf.Cos(theta));
    return D1(Xc, fi) * (h * c + D2(Xc, fi) * rho * Mathf.Pow(h, 2) * 0.5f) * (r - h * Mathf.Cos(theta));
}

private float TotalSideStress(float theta, float r){
    //Total side stress on the wheel.
    float b = wheelWidth;
    return r * b * LateralSheerStress(theta, r); // + BulldozingForce(theta);
}

private float IntegrateTrapezoidal(float a, float b, Func f, int nSteps, float r) {
    //Numerical integration of a function. Algorithm used is the trapezoidal method.
    if (a >= b)
        return -1;

    float h = (b - a) / nSteps;
    float result = (h / 2) * (f(a, r) + f(b, r));

    for (int i = 1; i < nSteps; i++)
        result += h * f(a + i * h, r);
return result;
}

private void deformNeighbourTerrain(Terrain terrain, int side, int distance, int heightPos, int heightSize, float sinkage) {
    // This function is used when there are two
    // neighbouring terrains and both are to be deformed
    // when the wheel is close to the edge between the
    // terrains.
    // This function will be removed as soon as the
    // custommade heightmap is implemented.
    if (side == 1) {
        float[,] heights = terrain.terrainData.GetHeights(
            heightPos, 0, heightSize, distance);
        for (int i = 0; i < distance; i++) {
            for (int j = 0; j < heightSize; j++) {
                // The shape of the deformation around the
                // collision point is a circle.
                if ((Mathf.Pow(i + (deformRadius - distance) + 1, 2) +
                    Mathf.Pow(j - deformRadius + 1, 2)) <= Mathf.Pow(
                        deformRadius + 1, 2)) {
                    heights[i, j] = heights[i, j] - sinkage;
                }
            }
        }
        terrain.terrainData.SetHeightsDelayLOD(heightPos, 0, heights);
    }
    else if (side == 2) {
        float[,] heights = terrain.terrainData.GetHeights(
            0, heightPos, distance, heightSize);
        for (int i = 0; i < heightSize; i++) {
            for (int j = 0; j < distance; j++) {
                // The shape of the deformation around the
                // collision point is a circle.
                if ((Mathf.Pow(i - deformRadius + 1, 2) +
                    Mathf.Pow(j + (deformRadius - distance) + 1, 2)) <= Mathf.Pow(
                        deformRadius + 1, 2)) {
                    heights[i, j] = heights[i, j] - sinkage;
                }
            }
        }
    }
}
terrain.terrainData.SetHeightsDelayLOD(0, heightPos, heights);
}

else if (side == 3) {
    float[,] heights = terrain.terrainData.GetHeights(
        heightPos, hmHeight - distance, heightSize, distance);

    for (int i = 0; i < distance; i++) {
        for (int j = 0; j < heightSize; j++) {
            // The shape of the deformation around the collision point is a circle.
            if ((Mathf.Pow(i - deformRadius, 2) + Mathf.Pow(j - deformRadius + 1, 2)) <= Mathf.Pow(deformRadius, 2)) {
                heights[i, j] = heights[i, j] - sinkage;
            }
        }
    }
    terrain.terrainData.SetHeightsDelayLOD(hmHeight - distance, heightPos, heights);
}

else if (side == 4) {
    float[,] heights = terrain.terrainData.GetHeights(
        hmWidth - distance, heightPos, distance, heightSize);

    for (int i = 0; i < heightSize; i++) {
        for (int j = 0; j < distance; j++) {
            // The shape of the deformation around the collision point is a circle.
            if ((Mathf.Pow(i - deformRadius, 2) + Mathf.Pow(j - deformRadius + 1, 2)) <= Mathf.Pow(deformRadius, 2)) {
                heights[i, j] = heights[i, j] - sinkage;
            }
        }
    }
    terrain.terrainData.SetHeightsDelayLOD(hmWidth - distance, heightPos, heights);
}

private bool NotInPrevDefPatterns(int i, int j, IntVec2[,] prevPos, IntVec2 currPos, IntVec2 circlePos, float deformRadius) {
    bool tempBool = true;

    for (int k = 3; k < 5; k++) {
tempBool = tempBool && ((Mathf.Pow(i - (circlePos.y - (currPos.y - prevPos[k].y)) + 1, 2) + Mathf.Pow(j - (circlePos.x - (currPos.x - prevPos[k].x)) + 1, 2)) > Mathf.Pow(deformRadius, 2));

return tempBool;

// This is used so that a function can be sent in as a argument to another function. Kinda like a function handle :D
private delegate float Func(float x, float r);

A.2 Script - tileController

using System;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

class Tile{
    // This is a class for the a tile. The tiles are the
    // plane objects that overlap the terrain at areas
    // where the terrain recently has been deformed.
    // The tiles are purely graphical and has nothing to do
    // with the interaction of the vehicle and the terrain.

    Terrain terrain;
    public GameObject plane;
    public IntVec2 position;
    public RenderTexture rt1;
    Material mat;
    Texture tex;
    int renderTextureRes = 1020;
    int tileSize = 10;
    int counter;
    public int [] neighbours;
    public RenderTexture [] neighbourTex;

    public Tile(GameObject p, IntVec2 pos){
        // TODO This only works if we have one terrain object
        // named "Terrain"
        terrain = GameObject.Find("Terrain").GetComponent<Terrain>();
        position = pos;
        plane = p;

        // The render texture is basically the heightmap of
        // the tile. The RT has only one channel which stores
        // the height
        // in every texel. The size of a tile is 10 x 10 m and
        // the resolution is 1020 x 1020 texels.
        rt1 = new RenderTexture(renderTextureRes,
                                renderTextureRes,0,RenderTextureFormat.RFloat);
        mat = plane.GetComponent<Renderer>().material;
        mat.shader = Shader.Find("Hidden/TemplateBlitShader");
        mat.renderQueue = 1;
        tex = Resources.Load("sand") as Texture;
        neighbours = new int[9];
        neighbours[1] = -1;}
neighbours[2] = -1;
neighbours[3] = -1;
neighbours[4] = -1;
neighbours[5] = -1;
neighbours[6] = -1;
neighbours[7] = -1;
neighbours[8] = -1;

neighbourTex = new RenderTexture[9];

rt1.enableRandomWrite = true;
rt1.Create();

Graphics.Blit(emptyTexture, rt1, mat);

mat.shader = Shader.Find("Nature/Terrain/Standard.");

// Set the properties of the terrain shader to the tile surface shader.
mat.SetFloat("_Metallic0", terrain.terrainData.
splatPrototypes[0].metallic);
//mat.SetFloat("_Metallic1", terrain.terrainData.
splatPrototypes[1].metallic);
//mat.SetFloat("_Metallic2", terrain.terrainData.
splatPrototypes[2].metallic);
//mat.SetFloat("_Metallic3", terrain.terrainData.
splatPrototypes[3].metallic);

mat.SetFloat("_Smoothness0", terrain.terrainData.
splatPrototypes[0].smoothness);
//mat.SetFloat("_Smoothness1", terrain.terrainData.
splatPrototypes[1].smoothness);
//mat.SetFloat("_Smoothness2", terrain.terrainData.
splatPrototypes[2].smoothness);
//mat.SetFloat("_Smoothness3", terrain.terrainData.
splatPrototypes[3].smoothness);

mat.SetTexture("_Splat0", terrain.terrainData.
splatPrototypes[0].texture);
//mat.SetTexture("_Splat1", terrain.terrainData.
splatPrototypes[1].texture);
//mat.SetTexture("_Splat2", terrain.terrainData.
splatPrototypes[2].texture);
//mat.SetTexture("_Splat3", terrain.terrainData.
splatPrototypes[3].texture);

mat.SetTexture("_Normal0", terrain.terrainData.
splatPrototypes[0].normalMap);
//mat.SetTexture("_Normal1", terrain.terrainData.
splatPrototypes[1].normalMap);
```csharp
int counter = 0;
}

public void SetTileMeshHeight(ComputeShader terrainDeformer)
{
    // This function reads the height values from the heightmap on the terrain object and writes it to a buffer,
    // which is then sent to the compute shader which writes to the render texture.
    // World position of the bottom left corner of the tile.
    Vector2 worldPos = new Vector2(position.x * (float)tileSize, position.y * (float)tileSize);

    // Heightmap width and height of the terrain.
    float hmWidth = terrain.terrainData.heightmapWidth;
    float hmHeight = terrain.terrainData.heightmapHeight;

    // Number of vertices per meter of the heightmap.
    float[] terrainNormalizeCoeff = new float[2];
    terrainNormalizeCoeff[0] = hmWidth / terrain.terrainData.size.x;
    terrainNormalizeCoeff[1] = hmHeight / terrain.terrainData.size.z;

    // The coordinates for the bottom left point on the tile in heightmap units.
    int coordx = (int)((worldPos.x - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
    int coordy = (int)((worldPos.y - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);

    // Array of height values
    float[,] bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int)(terrainNormalizeCoeff[0] * tileSize) + 1, (int)(terrainNormalizeCoeff[1] * tileSize) + 1);
    ComputeBuffer meshData = new ComputeBuffer(
        bufferArray.Length, sizeof(float));
    meshData.SetData(bufferArray);
}
// Set parameters and call the compute shader.
int kernelNum = terrainDeformer.FindKernel("SetHeights");
terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);
terrainDeformer.SetFloats("terrainRes",
    terrainNormalizeCoeff);
terrainDeformer.setTexture(kernelNum, "Result", rt1);
terrainDeformer.Dispatch(kernelNum, 1, renderTextureRes, 1);
meshData.Dispose();

// Some constants for the tile surface shader.
mat.SetInt("tileSize", tileSize);
mat.SetInt("tileNumX", (int)(worldPos.x/tileSize));
mat.SetInt("tileNumY", (int)(worldPos.y/tileSize));
mat.SetFloat("texTileSizeX", terrain.terrainData.
splatPrototypes[0].tileSize.x);
mat.SetFloat("texTileSizeY", terrain.terrainData.
splatPrototypes[0].tileSize.y);

public void DeformMesh(Vector2 WheelPosition, int diam,
        Vector2 wheelDir, float sinkage, ComputeShader terrainDeformer){
    // Deform the area around the wheel. Also meant to
    // apply physical deformations to the custommade
    // heightfield.
    // World position of the bottom left corner of the
tile.
    Vector2 worldPos = new Vector2(position.x * (float)
tileSize, position.y * (float) tileSize);
    // Heightmap width and height of the terrain.
    float hmWidth = terrain.terrainData.heightmapWidth;
    float hmHeight = terrain.terrainData.heightmapHeight;
    // Number of vertices per meter of the heightmap.
    float[] terrainNormalizeCoeff = new float[2];
    terrainNormalizeCoeff[0] = hmWidth / terrain.
terrainData.size.x;
    terrainNormalizeCoeff[1] = hmHeight / terrain.
terrainData.size.z;
    // The coordinates for the bottom left point on the
tile in heightmap units.
    int coordx = (int)((worldPos.x - terrain.gameObject.
        transform.position.x) * terrainNormalizeCoeff[0]);
int coory = (int) ((worldPos.y - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);

// Array of height values
float[,] bufferArray = terrain.terrainData.GetHeights(coordx, coory, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);

ComputeBuffer meshData = new ComputeBuffer(bufferArray.Length, sizeof(float));
meshData.SetData(bufferArray);

// Convert Vector2 arrays to int[] so that they can be sent in to the compute shader.
int[] pos = new int[2];
int[] offset = new int[2];
offset[0] = 0;
offset[1] = 0;
pos[0] = (int) WheelPosition.x;
pos[1] = (int) WheelPosition.y;
float[] wheelD = new float[2];
wheelD[0] = wheelDir.x;
wheeD[1] = wheelDir.y;

// Set parameters and call the compute shader.
int kernelNum = terrainDeformer.FindKernel("WriteTexture");
terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);
terrainDeformer.SetInt("diam", diam);
terrainDeformer.SetInts("pos", pos);
terrainDeformer.SetInts("offset", offset);
terrainDeformer.SetFloats("sc", wheelD);
terrainDeformer.SetFloat("sinkage", sinkage);
terrainDeformer.setTexture(kernelNum, "Result", rt1);
terrainDeformer.Dispatch(kernelNum, 1, diam, 1);

if (! (neighbours[1] == -1)) { // If the wheel is close enough to the north edge deformations on the northern tile is also occuring.

// Call the compute shader again for the neighbouring tile in order to deform it aswell.
coordx = (int) ((worldPos.x - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
coory = (int) ((worldPos.y + tileSize - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);

}
bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
meshData.SetData(bufferArray);
terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);

offset[0] = 0;
offset[1] = renderTextureRes;
terrainDeformer.SetInts("offset", offset);
terrainDeformer.SetTexture(0, "Result", neighbourTex[1]);
terrainDeformer.Dispatch(kernelNum, 1, diam, 1);
}

if (!(neighbours[2] == -1)){ // If the wheel is close enough to the northeastern edge deformations on the northeastern tile is also occurring.
  // Call the compute shader again for the neighbouring tile in order to deform it as well.
  coordx = (int) ((worldPos.x + tileSize - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
  coordy = (int) ((worldPos.y + tileSize - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);
  bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
  meshData.SetData(bufferArray);
terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);
  offset[0] = renderTextureRes;
  offset[1] = renderTextureRes;
  terrainDeformer.SetInts("offset", offset);
  terrainDeformer.SetTexture(0, "Result", neighbourTex[2]);
  terrainDeformer.Dispatch(kernelNum, 1, diam, 1);
}

if (!(neighbours[3] == -1)){ // If the wheel is close enough to the eastern edge deformations on the eastern tile is also occurring.
  // Call the compute shader again for the neighbouring tile in order to deform it as well.
coordx = (int) ((worldPos.x + tileSize - terrain.
gameObject.transform.position.x) *
terrainNormalizeCoeff[0]);
coordy = (int) ((worldPos.y - terrain.gameObject.
transform.position.y) * terrainNormalizeCoeff
[1]);
bufferArray = terrain.terrainData.GetHeights(coordx
, coordy, (int) (terrainNormalizeCoeff[0] *
tileSize) + 1, (int) (terrainNormalizeCoeff[1] *
tileSize) + 1);
meshData.SetData(bufferArray);
terrainDeformer.SetBuffer(kernelNum, "meshData",
meshData);

offset[0] = renderTextureRes;
offset[1] = 0;
terrainDeformer.SetInts("offset", offset);
terrainDeformer.SetTexture(0,"Result",neighbourTex
[3]);
terrainDeformer.Dispatch(kernelNum,1,diam,1);
} 

if (!(neighbours[4] == -1)){ //If the wheel is close
  enough to the southeast edge deformations on the
  southeastern tile is also occurring.

  //Call the compute shader again for the
  neighbouring tile in order to deform it aswell.
  coordx = (int) ((worldPos.x + tileSize - terrain.
  gameObject.transform.position.x) *
  terrainNormalizeCoeff[0]);
  coordy = (int) ((worldPos.y - tileSize - terrain.
  gameObject.transform.position.y) *
  terrainNormalizeCoeff[1]);
  bufferArray = terrain.terrainData.GetHeights(coordx
  , coordy, (int) (terrainNormalizeCoeff[0] *
  tileSize) + 1, (int) (terrainNormalizeCoeff[1] *
  tileSize) + 1);
  meshData.SetData(bufferArray);
  terrainDeformer.SetBuffer(kernelNum,"meshData",
  meshData);

  offset[0] = renderTextureRes;
  offset[1] = -renderTextureRes;
  terrainDeformer.SetInts("offset", offset);
  terrainDeformer.SetTexture(0,"Result",neighbourTex
  [4]);
  terrainDeformer.Dispatch(kernelNum,1,diam,1);
}
if (!(neighbours[5] == -1)){ //If the wheel is close enough to the south edge deformations on the southern tile is also occuring.
    //Call the compute shader again for the neighbouring tile in order to deform it aswell.
    coordx = (int) ((worldPos.x - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
    coordy = (int) ((worldPos.y - tileSize - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);
    bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
    meshData.SetData(bufferArray);
    terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);
    offset[0] = 0;
    offset[1] = -renderTextureRes;
    terrainDeformer.SetInts("offset", offset);
    terrainDeformer.SetTexture(0,"Result", neighbourTex[5]);
    terrainDeformer.Dispatch(kernelNum, 1, diam, 1);
}

if (!(neighbours[6] == -1)){ //If the wheel is close enough to the southwest edge deformations on the southwestern tile is also occuring.
    //Call the compute shader again for the neighbouring tile in order to deform it aswell.
    coordx = (int) ((worldPos.x - tileSize - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
    coordy = (int) ((worldPos.y - tileSize - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);
    bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
    meshData.SetData(bufferArray);
    terrainDeformer.SetBuffer(kernelNum, "meshData", meshData);
    offset[0] = -renderTextureRes;
    offset[1] = -renderTextureRes;
terrainDeformer.SetInts("offset", offset);
terrainDeformer.setTexture(0,"Result", neighbourTex[6]);
terrainDeformer.Dispatch(kernelNum,1,diam,1);
}

if (!(neighbours[7] == -1)) { // If the wheel is close enough to the west edge deformations on the western tile is also occurring.
    // Call the compute shader again for the neighbouring tile in order to deform it as well.
    coordx = (int) ((worldPos.x - tileSize - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
    coordy = (int) ((worldPos.y - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);
    bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
    meshData.SetData(bufferArray);
    terrainDeformer.SetBuffer(kernelNum,"meshData", meshData);
    offset[0] = -renderTextureRes;
    offset[1] = 0;
    terrainDeformer.SetInts("offset", offset);
    terrainDeformer.setTexture(0,"Result", neighbourTex[7]);
    terrainDeformer.Dispatch(kernelNum,1,diam,1);
}

if (!(neighbours[8] == -1)) { // If the wheel is close enough to the northwest edge deformations on the northwestern tile is also occurring.
    // Call the compute shader again for the neighbouring tile in order to deform it as well.
    coordx = (int) ((worldPos.x - tileSize - terrain.gameObject.transform.position.x) * terrainNormalizeCoeff[0]);
    coordy = (int) ((worldPos.y + tileSize - terrain.gameObject.transform.position.y) * terrainNormalizeCoeff[1]);
    bufferArray = terrain.terrainData.GetHeights(coordx, coordy, (int) (terrainNormalizeCoeff[0] * tileSize) + 1, (int) (terrainNormalizeCoeff[1] * tileSize) + 1);
meshData.SetData(bufferArray);
terrainDeformer.SetBuffer(kernelNum,"meshData",
meshData);

offset[0] = -renderTextureRes;
offset[1] = renderTextureRes;
terrainDeformer.SetInts("offset", offset);
terrainDeformer.SetTexture(0,"Result",neighbourTex
[8]);
terrainDeformer.Dispatch(kernelNum,1,diam,1);
}

meshData.Dispose();
}

public void ResetThisTile()
{
    // Resets the rt on this tile.
    mat.shader = Shader.Find("Hidden/ResetShader");
    Graphics.Blit(emptyTexture,rt1,mat);

    mat.shader = Shader.Find("Nature/Terrain/Standard.");

    mat.SetFloat("
    _Metallic0", terrain.terrainData.
    splatPrototypes[0].metallic);
    //mat.SetFloat("_Metallic1", terrain.terrainData.
    splatPrototypes[1].metallic);
    //mat.SetFloat("_Metallic2", terrain.terrainData.
    splatPrototypes[2].metallic);
    //mat.SetFloat("_Metallic3", terrain.terrainData.
    splatPrototypes[3].metallic);

    mat.SetFloat("_Smoothness0", terrain.terrainData.
    splatPrototypes[0].smoothness);
    //mat.SetFloat("_Smoothness1", terrain.terrainData.
    splatPrototypes[1].smoothness);
    //mat.SetFloat("_Smoothness2", terrain.terrainData.
    splatPrototypes[2].smoothness);
    //mat.SetFloat("_Smoothness3", terrain.terrainData.
    splatPrototypes[3].smoothness);

    mat.SetTexture("_Splat0", terrain.terrainData.
    splatPrototypes[0].texture);
    //mat.SetTexture("_Splat1", terrain.terrainData.
    splatPrototypes[1].texture);
    //mat.SetTexture("_Splat2", terrain.terrainData.
    splatPrototypes[2].texture);
    //mat.SetTexture("_Splat3", terrain.terrainData.
    splatPrototypes[3].texture);
mat.SetTexture("_Normal0", terrain.terrainData.
splatPrototypes[0].normalMap);
    //mat.SetTexture("_Normal1", terrain.terrainData.
splatPrototypes[1].normalMap);
    //mat.SetTexture("_Normal2", terrain.terrainData.
splatPrototypes[2].normalMap);
    //mat.SetTexture("_Normal3", terrain.terrainData.
splatPrototypes[3].normalMap);

    //mat.shader = Shader.Find("Custom/SurfaceShader");
    mat.SetTexture("_RealTex", tex);
    mat.renderQueue = 1;
}   
   
private int GetLinearIndex(IntVec2 posIndex, int length)
{
    return (posIndex.x + posIndex.y * length);
}   

public class tileController : MonoBehaviour {
    //This class keeps track of which tile is in what
    //position and has a queue for the tiles.

    Terrain terrain;
    Tile[] planes;
    GameObject plane;
    int currentTileNum, firstInQueue;
    List<int> tileList;
    Hashtable ht;
    public ComputeShader terrainDeformer;
    ComputeBuffer noiseData;
    Vector2[,] randomGradients;
    int tileSize = 10;
    int noiseResolution = 100;

    //This Dictionary is made to look up which tile number
    //is on a given position. The position is a IntVec2
    //in order to convert two ints to one float in a unique
    //manner one of the floats is multiplied by a big
    //number, 1000 in this case, and the other by a small number,
    //0.01 in this case. The only way this is not unique
    //is if
    //the y position is bigger than 1000/0.01 which will
    //never happen. This can also be achieved with
    //primenumbers but I CBA.
    Dictionary<float, int> positionLookUp;
    Vector3 origin;
public void InitializeTileBuffer(int numTiles) {

    //TODO, make this get a reference to the terrain object in a more general manner, this only works if there is one terrain.
    terrain = GameObject.Find("Terrain").GetComponent<Terrain>();
    planes = new Tile[numTiles];
    tileList = new List<int>(numTiles);
    ht = new Hashtable(numTiles);
    positionLookUp = new Dictionary<float, int>(numTiles);
    origin = terrain.GetPosition();
    plane = Resources.Load("plane") as GameObject;
    firstInQueue = 0;
    terrainDeformer = Resources.Load("customTerrainDeformer") as ComputeShader;

    //Random gradients are the random pattern that is used for the perlin noise in the compute shader.
    randomGradients = new Vector2[noiseResolution + 1, noiseResolution + 1];
    for (int i = 0; i < noiseResolution; i++) {
        for (int j = 0; j < noiseResolution; j++) {
            randomGradients[i, j] = new Vector2(UnityEngine.Random.value * 2 - 1, UnityEngine.Random.value * 2 - 1);
            randomGradients[i, j] = randomGradients[i, j] / (Mathf.Sqrt(randomGradients[i, j].x * randomGradients[i, j].x + randomGradients[i, j].y * randomGradients[i, j].y));
        }
    }
    for (int i = 0; i < noiseResolution + 1; i++) {
        randomGradients[i, noiseResolution] = new Vector2(randomGradients[i, 0].x, randomGradients[i, 0].y);
        randomGradients[noiseResolution, i] = new Vector2(randomGradients[0, i].x, randomGradients[0, i].y);
    }

    noiseData = new ComputeBuffer(randomGradients.Length, 2 * sizeof(float));
    noiseData.SetData(randomGradients);
    terrainDeformer.SetBuffer(terrainDeformer.FindKernel("WriteTexture"), "noiseData", noiseData);
    terrainDeformer.SetInt("noiseResolution", noiseResolution);

    //Instantiate the tiles.
    for (int i = numTiles - 1; i > -1; i--){
        tileList.Add(i);
    }
}
positionLookUp.Add(i*1000,i);
planes[i] = new Tile((GameObject) Instantiate(plane
, new Vector3(origin.x + (i)*tileSize + 5,
origin.y + 2, origin.z + 5), Quaternion.identity
), new IntVec2(i,0));
planes[i].position = new IntVec2(i,0);
}

public void OnDestroy(){
    // This is never called thus noiseData is never
    // disposed, which is what causes the memory leak...
    // I think
    noiseData.Dispose();
}

public void DeformTerrain(Vector2 wheelPosition, float
sinkage, int deformDiam, Vector2 wheelDir){
    planes[currentTileNum].DeformMesh(wheelPosition,
    deformDiam, wheelDir, sinkage, terrainDeformer);
}

public void ChangeTile(IntVec2 pos, int side){
    // This function handles the tile queue, that is which
    // tile is going to be reset and put in a new
    // position.
    // The principle is that when changing a tile the one
    // that was visited last is changed.
    int tileNum;

    // This if statement checks if the position the wheel
    // is on does not have a tile on it.
    if (!positionLookUp.TryGetValue(pos.x * 1000 + pos.y
    * 0.01f, out tileNum)){
        // If the queue does not contain the tile that moved
        // last time, the tile ID has to be added last in
        // the queue.
        if (!tileList.Contains(firstInQueue)){
            tileList.Add(firstInQueue);
        }

        // Get the ID of the tile first in the queue and
        // remove it from the queue.
        firstInQueue = tileList[0];
        tileList.RemoveAt(0);

        // Remove the tile from the position lookup table.
    }
}
positionLookUp.Remove(planes[firstInQueue].position.x * 1000 + planes[firstInQueue].position.y * 0.01f);

//planes[firstInQueue].ResetThisTile();

//Change the actual position of the tile.
planes[firstInQueue].plane.transform.position = new Vector3(origin.x + (pos.x)*tileSize + 5, origin.y + 2, (pos.y)*tileSize + origin.z + 5);

//Change the position of the tile.
planes[firstInQueue].position = new IntVec2(pos.x, pos.y);

//Add the tile to the position lookup table.
positionLookUp.Add(pos.x * 1000 + pos.y * 0.01f, firstInQueue);

//Side is the side on which this tile is on relative to the current tile.
planes[currentTileNum].neighbours[side] = firstInQueue;
planes[currentTileNum].neighbourTex[side] = planes[firstInQueue].rt1;

//Set the height of the tile.
planes[firstInQueue].SetTileMeshHeight(terrainDeformer);
}

public void ChangeCurrentTile(IntVec2 pos){
    //Keeps track of the neighbours of a tile at a specific position.
    int tileNum;

    //Look if there is a tile on the position pos. If there is, tileNum is the ID of that tile.
    if (positionLookUp.TryGetValue(pos.x * 1000 + pos.y * 0.01f, out tileNum)){
        //neighbours[X] returns the ID of the neighbour at side X.
        //1 − north, 2 − northeast, 3 − east, 4 − southeast, 5 − south, 6 − southwest, 7 − west, 8 − northwest
        //−1 means no neighbour
// TODO implement enum for this if you want. I prefer to just use 1 as north and then rotate clockwise.
planes[currentTileNum].neighbours[1] = -1;
planes[currentTileNum].neighbours[2] = -1;
planes[currentTileNum].neighbours[3] = -1;
planes[currentTileNum].neighbours[4] = -1;
planes[currentTileNum].neighbours[5] = -1;
planes[currentTileNum].neighbours[6] = -1;
planes[currentTileNum].neighbours[7] = -1;
planes[currentTileNum].neighbours[8] = -1;

currentTileNum = tileNum;

// Remove and add the current tile so that it is put last in the queue that keeps track of which tile
// was visited last.
tileList.Remove(tileNum);
tileList.Add(tileNum);

// Check if the new current tile has a neighbour to the...
// North
if (positionLookUp.TryGetValue((pos.x * 1000 + (pos.y + 1) * 0.01f, out tileNum))
planes[currentTileNum].neighbours[1] = tileNum;
planes[currentTileNum].neighbourTex[1] = planes[tileNum].rt1;
}

// Northeast
if (positionLookUp.TryGetValue((pos.x + 1) * 1000 + (pos.y + 1) * 0.01f, out tileNum))
planes[currentTileNum].neighbours[2] = tileNum;
}

// East
if (positionLookUp.TryGetValue((pos.x + 1) * 1000 + pos.y * 0.01f, out tileNum))
planes[currentTileNum].neighbours[3] = tileNum;
}

// Southeast
if (positionLookUp.TryGetValue((pos.x + 1) * 1000 + (pos.y - 1) * 0.01f, out tileNum))

```csharp
planes[currentTileNum].neighbours[4] = tileNum;

// South
if (positionLookUp.TryGetValue((pos.x * 1000 + (pos.y - 1) * 0.01f, out tileNum)){
    planes[currentTileNum].neighbours[5] = tileNum;
}

// Southwest
if (positionLookUp.TryGetValue((pos.x - 1) * 1000 + (pos.y - 1) * 0.01f, out tileNum)){
    planes[currentTileNum].neighbours[6] = tileNum;
}

// West
if (positionLookUp.TryGetValue((pos.x - 1) * 1000 + pos.y * 0.01f, out tileNum)){
    planes[currentTileNum].neighbours[7] = tileNum;
    planes[currentTileNum].neighbourTex[7] = planes[tileNum].rt1;
}

// Northwest
if (positionLookUp.TryGetValue((pos.x - 1) * 1000 + (pos.y + 1) * 0.01f, out tileNum)){
    planes[currentTileNum].neighbours[8] = tileNum;
    planes[currentTileNum].neighbourTex[8] = planes[tileNum].rt1;
}
```
A.3 Script - Density

using UnityEngine;
using System.Collections;

public class Density : MonoBehaviour {
    [SerializeField] public Terrain west, east, north, south;
    public float density = 1600f;
    public float cohMod = 1.71f;
    public float friMod = 1000f;
    public float defExp = 1f;
    public float cohStress = 800f;
    public float kappa = 0.7f;
    public float friAngle = 0.5f;
    public float sheerDefModulus = 1f;
    public float diggingIndex = 0.00001f;
}
### A.4 Compute shader - customTerrainDeformer

```cpp
#pragma kernel WriteTexture
#pragma kernel SetHeights
#pragma target 3.0

RWTexture2D<float> Result;
int diam;
int2 pos;
int2 offset;
int hmOffset;
float2 sc;
float sinkage;
float originalHeight;
int noiseResolution;

float2 terrainRes;
StructuredBuffer<float> meshData;
StructuredBuffer<float2> noiseData;

float FadeFunction(float t){
    // Fade function as defined by Ken Perlin. This eases coordinate values
    // so that they will ease towards integral values. This ends up smoothing
    // the final output.
    // 6t^5 - 15t^4 + 10t^3

    return t * t * t * (t * (t * 6 - 15) + 10);
}

float Pnoise(float x, float y, int2 posOnRT, float originalHeight, float currHeight){
    // Perlin noise function, this noise is applied to the deformations.

    float maxNoise = 0.01;

    float g1 = dot(noiseData[1, posOnRT.y * noiseResolution + posOnRT.x], float2(x,y));
    float g2 = dot(noiseData[1, posOnRT.x + 1 + posOnRT.y * noiseResolution], float2(1 - x,y));
    float g3 = dot(noiseData[1, posOnRT.x + (posOnRT.y + 1) * noiseResolution], float2(x,1 - y));
    float g4 = dot(noiseData[1, posOnRT.x + 1 + (posOnRT.y + 1) * noiseResolution], float2(1 - x,1 - y));

    float u = FadeFunction(x);
    float v = FadeFunction(y);
```
float noise = (0.08 * lerp(lerp(g1, g2, u), lerp(g3, g4, u), v));
return min(noise * (originalHeight - currHeight), maxNoise);
//return noise * step(originalHeight + 0.1, currHeight);
}

float Height(float height, float x, float y, float originalHeight)
//This function calculates the graphical deformation heights
float texelSize = 0.0009804; // 1 / renderTextureRes
float tsw = texelSize * 10; // plane width is 10
float sigma = 12;

//sc is the vector that is pointing in the direction that the vehicle is travelling in,
//described in world coordinates.
//sc.x = sin(a), sc.y = cos(a). a is the angle between the world z axis and the traveling direction.

//This equation makes the terrain go back to its original value right before its being deformed by the wheel.
height += exp(-(x - sc.x*50)*(x - sc.x*50) + (y - sc.y *50)*(y - sc.y*50))/(2*sigma*sigma) * (300 * (originalHeight - 0.02) - height) * 0.7;

//note that only the first factor (the polynomial factor) of the function needs to change coordinates
//since the exp factor is uniform in all directions (the exp term is just a 2d gaussian clock)
height -= 4 * (sigma * sigma - (sc.y * x - sc.x * y) * (sc.y * x - sc.x * y)) * exp(-(x*x + y*y)/(2 * sigma *sigma)) * sinkage / (sigma*sigma);
return height;
}

float BilinearInterpolation(float q11, float q12, float q21, float q22, float x, float y)
{
//q11, q12, q21 and q22 are heights at corners. x and y are coordinates where in the quad
//the point that is to be interpolated is.
float fxy1 = (1 - x)*q11 + x*q21;
float fxy2 = (1 - x)*q12 + x*q22;

return (1 - y)*fxy1 + y*fxy2;
}

/*float4x4 CoeffMatrix(float4x4 f) {
 float4x4 aij = mul(mul(float4x4
 (1,0,0,0,0,0,1,0,
 (1,0,-3,2,0,0,3,-2,0,1,-2,1,0,0,-1,1)))
 */

float BicubicInterpolation(float4x4 aij, float x, float y ) {
 return aij[0,0] + aij[1,0]*x + aij[2,0]*x*x + aij[3,0]*
 x*x*x + aij[0,1]*y + aij[1,1]*x*y + aij[2,1]*x*x*y +
 aij[3,1]*x*x*x*y + aij[0,2]*y*y + aij[1,2]*x*y*y +
 aij[2,2]*x*x*y*y + aij[3,2]*x*x*x*y*y + aij[0,3]*y*y *
y + aij[1,3]*x*y*y*y + aij[2,3]*x*x*y*y*y + aij
[3,3]*x*x*x*y*y*y;
}

// 200 is the size of the square centered on the wheel that is being deformed.
[numthreads(200,1,1)]
void WriteTexture (uint3 id : SV_DispatchThreadID)
{
    // This function is used when a tile is being deformed.
    // It essentially calculates new heights for a small
    // area around the wheel and writes this to the
    // render texture.

    float texelSize = 1 / 1020; // 1 / renderTextureRes
    float tsw = texelSize * 10; // plane width is 10
    float sigma = 12;
    float x = float(id.x) - 100.0;
    float y = float(id.y) - 100.0;
    int2 posOnRT = int2(id.x, id.y) + pos - int2(100,100) -
    offset;

    float heightmapCoordx = float(posOnRT.x) * 100 / 1020;
    float heightmapCoordy = float(posOnRT.y) * 100 / 1020;
    float noiseCoordx = float(id.x) * float(noiseResolution
    ) / 200.0;
    float noiseCoordy = float(id.y) * float(noiseResolution
    ) / 200.0;

    float f00 = meshData[1, (int(heightmapCoordy)) * 101 +
    int(heightmapCoordx) ];
float f01 = meshData[1, (int)(heightmapCoordy) + 1] * 101 + int(heightmapCoordx)];
float f10 = meshData[1, (int)(heightmapCoordy)) * 101 + int(heightmapCoordx) + 1];
float f11 = meshData[1, (int)(heightmapCoordy) + 1] * 101 + int(heightmapCoordx) + 1];

float originalHeight = BilinearInterpolation(f00, f01, f10, f11, heightmapCoordx - floor(heightmapCoordx), heightmapCoordy - floor(heightmapCoordy));

float height = Result[posOnRT];
height = Height(height, x, y, originalHeight);
//int2(int(heightmapCoordx),int(heightmapCoordy))
Result[posOnRT] = height + Pnoise(noiseCoordx - floor(noiseCoordx), noiseCoordy - floor(noiseCoordy), int2(int(noiseCoordx), int(noiseCoordy), 300 * (originalHeight - 0.02), height));
}

//The number 1020 is the resolution of the render texture
[numthreads(1020,1,1)]
void SetHeights (uint3 id : SV DispatchThreadID)
{
    //This function is called when a tile is moved to a new
    //position. It reads the heightdata from a buffer and
    //sets it to the render texture.
    //Currently meshData is a buffer with heightdata from
    //the heightmap of the terrain object, but the
    //intention is to use the custommade
    //heightmap (which Jens created) for this.

    //x and y is the indecies of the meshData buffer.
    float x = float(id.x) * 100 / 1020;
    float y = float(id.y) * 100 / 1020;

    //The point of all the height values and derivatives is
    //to use bicubic interpolation.
    //However currently bilinear interpolation is used.

    float f_1_1 = meshData[1, (int)(y) - 1] * 101 + int(x) - 1];
    float f_10 = meshData[1, (int)(y)) * 101 + int(x) - 1];
    float f_11 = meshData[1, (int)(y) + 1] * 101 + int(x) - 1];
    float f_12 = meshData[1, (int)(y) - 1] * 101 + int(x) + 2];
float f1_1 = meshData[1, (int)(y) - 1] * 101 + int(x) +
1];
float f1_10 = meshData[1, (int)(y)] * 101 + int(x) + 1];
float f1_11 = meshData[1, (int)(y) + 1] * 101 + int(x) +
1];
float f1_12 = meshData[1, (int)(y) + 2] * 101 + int(x) +
1];

float f0_1 = meshData[1, (int)(y) - 1] * 101 + int(x)];
float f0_0 = meshData[1, (int)(y)] * 101 + int(x)];
float f0_10 = meshData[1, (int)(y) + 1] * 101 + int(x)];
float f0_2 = meshData[1, (int)(y) + 2] * 101 + int(x)];

float f2_1 = meshData[1, (int)(y) - 1] * 101 + int(x) +
2];
float f2_0 = meshData[1, (int)(y)] * 101 + int(x) + 2];
float f2_10 = meshData[1, (int)(y) + 1] * 101 + int(x) +
2];
float f2_11 = meshData[1, (int)(y) + 2] * 101 + int(x) +
2];

float fx00 = ((f10 - f00) + (f00 - f10))/2;
float fx10 = ((f20 - f10) + (f10 - f00))/2;
float fx01 = ((f11 - f01) + (f01 - f11))/2;
float fx11 = ((f21 - f11) + (f11 - f01))/2;

float fy_10 = ((f11 - f_10) + (f_10 - f1_1))/2;
float fy00 = ((f01 - f00) + (f00 - f0_1))/2;
float fy10 = ((f11 - f10) + (f10 - f1_1))/2;
float fy20 = ((f11 - f20) + (f20 - f2_1))/2;
float fy_11 = ((f11 - f_10) + (f_10 - f_11))/2;
float fy01 = ((f02 - f01) + (f01 - f00))/2;
float fy11 = ((f12 - f11) + (f11 - f10))/2;
float fy21 = ((f22 - f21) + (f21 - f20))/2;

float fxy00 = ((f10 - f00) + (f00 - f_10))/2;
float fxy10 = ((f20 - f10) + (f10 - f00))/2;
float fxy01 = ((f11 - f01) + (f01 - f_11))/2;
float fxy11 = ((f21 - f11) + (f11 - f01))/2;

Result[id. xy] = 300 * (BilinearInterpolation(f00, f01, f10, f11, x - floor(x), y - floor(y)) - 0.02);
References


http://web.mit.edu/senator/www/docs/Multipass_Senatore_Sandu.pdf


http://link.springer.com/article/10.1007%2Fs00603-012-0281-7#page-1