**Investigating a real-time Hydraulic Erosion Simulation to be used for terrain generation in games**

1 Introduction

This portfolio project will contain an investigation into real-time hydraulic erosion when creating realistic terrain in video games and a program that will use techniques found in the investigation to create a real-time hydraulic erosion simulation. Terrain is one of the most important aspects of emulating a realistic virtual experience whether it’s in a computer game that requires a large-scale terrain for an open world experience, a movie that requires a fantasy style location that might be too dangerous to film in or a training simulation that requires the most realistic environment to help train people.

1.1 Aims and Goals

What needs to be achieved

* Aims – quite broad
* Objectives – very specific

2 Literature Review

2.1 Terrain

(INTRO CAN BE FINISHED LATER)

Generating realistic terrain for games is a long and time-consuming process. To solve this problem developers have found quicker and more effective ways for generating realistic terrain. When talking about terrain generation the first step is how that data is represented. Terrain data can be represented with two main models. These being a volumetric model and an elevation model.

2.1.1 Terrain Representation

The elevation model can be described with either an elevation function or a discrete heightfield. The most commonly used method is the discrete heightfield. The discrete heightfield uses a two-dimensional grid to represent the altitude of each position. This means that it is unable to recreate suspended materials like overhangs, caves, and arches. But it is a lot less data heavy which allows a simulation to be ran a lot faster with more area to work with. Elevation functions use a formula that can generate a point of altitude at any point in terrain. This allows for a program to generate multiple points to create a terrain of any size required.[2]

The volumetric model is similar to the elevation model but instead of using two-dimensional space it uses voxels which allow for a three-dimensional space where each cell represents a material at a particular position. This allows for the use of features like overhangs, caves, and arches. Volumetric models are very data heavy which makes them slower to run. There are ways to optimize the data structure by using compression techniques like Sparse Voxel Octrees to reduce the memory cost. This is done using an octree. An octree is where a three-dimensional space gets recursively divided into subspaces of children nodes until each voxel only contains one point or multiple point of similar data.[1] This technique can be used in open world games which require real time interaction.

2.1.2 Terrain Generation

When creating terrain there are two main ways for acquiring the terrain data. These being data sets and procedural generation. Data sets can be created by artists modelling the terrain in a 3D modelling software or by collecting data from the real world. There are also ways for users to input a small amount of detail into a program which then outputs a model based on the users’ input. Procedural generation is a method where programs generate their own data with no user input to be used for terrain. These methods include Perlin or Simplex noise, Cellular Automata and Diamond Square[17].

Noise is essentially a function that generates a set of random numbers. It is used in multiple different procedural generation methods. But random numbers on their own aren’t what you want when modelling terrain. What a noise function does is it takes white noise, which is a source of random uniformly distributed numbers with no correlation whatsoever between successive numbers and gives it a structure or pattern to then be used. White noise can be generated by a random physical process, such as the thermal noise that occurs within many analogue electronic systems.[7]

There are multiple different functions that can be used with handling noise. One famous function is the Perlin noise function discovered by Ken Perlin in 1983[8] which was then improved in 2001 by Ken Perlin again who called it simplex noise[9].

Fractals are defined as sets that exhibit self-similarity under scaling. This means when you zoom in on a fractal pattern, the same pattern will repeat itself over and over again. There are many different fractal algorithms that have been discovered each with their own unique properties and characteristics. Fractals can be described by using a property called a fractal dimension. Fractal dimensions allow for real number to be used to describe a dimension. For example, 2.3, 2 being the second dimension and .3 being the fractal increment. The fractal increment determines the roughness of the fractal with 0 being flat and .999 being rough. The equation for finding a fractal dimension was discovered by Benoit Mandelbrot.[3] This equation being:

D = log(N) / log(1 / r)

Where D is the fractal dimension, N is the amount an object can be divided and r being the ratio of how much the object has been divided. A good example to show this would be the von Koch snowflake curve which was known as an “early mathematical monster” where a simple line segment is divided into thirds and the middle third is replaced by two segments forming part of an equilateral triangle. At the next stage in the construction each of these 4 segments are replaced by 4 new segments with length 1/3 of their parent according to the original pattern. This then is repeated over and over again yielding the von Koch curve.

(PICTURE OF THE SNOWFLAKE CURVE)

So, with the amount the object gets divided being 4, each of these section is scaled down by 1/3. The equation for the von Koch snowflake curve would be D = log(4) / log(3). This means that the fractal dimension equals 1.26. There are a few different ways these fractals can be used to generate terrain. One of them being Multifractals. What this does is it takes a Noise function adds an offset to it and then multiplies it by the power of the lacunarity which is the change in scale between successive levels of detail and the fractal dimension. It then repeats this over a given number of octaves which is the number of scales at which you’re adding in smaller details.[5][6]

2.2 Water/Fluid Simulation

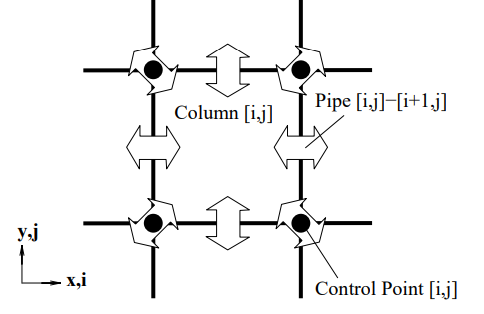
Most fluids in animation are governed by the Navier-Stokes equations. These equations are a set of partial differential equations that are held throughout the fluid. The Navier-Stokes equations are split into two equations. One being the momentum equation and the other being the incompressibility equation. The momentum equation is derived from newtons second law where mass times acceleration equals force. The momentum equation has been adapted for flow. This is done by including viscosity, pressure and velocity force related to time. The equation looks like this:

The p is the density of the fluid, du/dt is the time dependent velocity, P being the pressure, v being the dynamic viscosity and F being any external forces interacting with the fluid which in most cases is usually gravity. The incompressibility equation is to state that the continuity does not change, and that mass is conserved. The equation looks like this:

The first part of this equation is the divergence, and the u is the velocity vector field. This equation describes how the mass of a fluid across the field must never change therefor has to be zero.[10][11]

Now that we have talked about the Navier-Stokes equations we can talk about the methods used to simulate water. When simulating water there are two main approaches to tracking the fluid. These being the Eularian approach and the Lagrangian approach. The Lagrangian approach, which was named after the mathematician Lagrange, labels each point in a fluid as a separate particle with its own position and velocity. This is well known as a particle system. One of the most common particle systems is called Smooth Particle Hydrodynamics which was discovered by R.A.Gingold and J.J.Monaghan in 1977[12]. The Eularian approach, which was named after the mathematician Euler, takes a different approach. Instead of tracking each particle it tracks a fixed point in space and tracks the fluid density and pressure of that fixed position. This is also known as a grid-based approach. There are multiple grid-based methods used when simulating fluids. One of these being the Pipe Model method which was discovered by James F. O’Brien and Jessica K. Hodgins[14] and was then later adapted to run faster on a Graphics Processing Unit (GPU) by Xing Mei, Philippe Decaudin and Bao-Gang Hu. There are of course many more.[15]

Smooth Particle Hydrodynamics is one of the most commonly used particle-based simulation methods. It works by first identifying the properties of each particle. It does this by acquiring all the neighbouring particles within the kernel radius. Computes the kernel weight at the centres of the neighbouring particles. Finds the weight of the physical quantity carried by the neighbouring particles by the kernel values and then sums the values obtained. This method then uses a simplified version of the Navier-Stokes equation to calculate the motion.[13]

The Pipe Model method is a grid-based water simulation method that uses virtual pipes that connect neighbouring cells to transfer water between them. It stores its data on a two-dimensional grid where the height of the water is stored in each position on the grid. Each cell is connected eight ways to every neighbouring cell and looks like this:

Picture from [14]

To transfer water, it calculates the pressure between each pipe to measure which direction the water will flow. The equation to determine the flow in the pipes are derived from the physical laws of hydrostatic pressure. The static pressure of the grid cell at position [i, j] would be:

Where is the density of the fluid, g is the acceleration due to gravity, is the set atmospheric pressure in the system, is external forces that don’t include gravity like objects hitting the surface of the water, and is the height of the grid cell at [i, j]. To then solve the acceleration of a pipe it uses an equation that’s derived from Newtons second law being F = ma. Which looks like:

Where is the pipe connecting a grid cell to a neighbouring grid cell, c is the cross-sectional area of the pipe, and m is the mass of the fluid in the pipe. The mass of the fluid in the pipe can be expressed with , where l is the length of the pipe. Assuming that the set atmospheric pressure is a constant this equation can then be rewritten as:

To calculate the flow in the pipe it then assumes that the acceleration over a time step is constant which then gives us the equation:

To then calculate the net volume change of a time step it takes the flow of each pipe connected to the current grid cell, adds it to the previous time step flow and divides it by two. Doing that gives us this equation:

After this at the end of each time step all grid cells are tested to see if any are at a negative volume. If so, then all the pipes that are removing water are scaled back. This is repeated until all grid cell volumes are positive[14]. This method then got adapted to run on a GPU by Xing Mei, Philippe Decaudin and Bao-Gang Hu[15]. The changes they made to this method include adding a flux variable to each grid cell which are used to hold the amount of fluid that will be transferred from the current cell to the allocated neighbouring cell. They then adapt the equations around the flux. This allows for this method to be parallelized on a GPU.

The Shallow Water equations are a set of partial differential equations that describe the behaviour of water waves in shallow areas of water. These equations are derived from the Navier-Stoke equations and make a few assumptions about the simulated water. One assumption being that the horizontal length of the water is greater than the vertical length. It also assumes that the vertical velocities of the water are very minimal. The shallow water equations are used in multiple grid-based methods to simulate the surface of the water as grid-based methods find it hard to simulate the surface accurately.

When it comes to how efficient a water simulation is, both particle-based and grid-based methods have their own advantages and disadvantages. Particle-based methods only require for you to simulate where water is precent, while using a grid-based method requires the entire world space to be simulated at all time steps. But particle-based methods find difficulties with large bodies of water as it would require thousands of particles to be simulated each time step. This shows that particle-based methods are better suited to small bodies of water whilst grid-based methods are more efficient at running larger bodies of water such as rivers, canals, or oceans/waves. This has led to people investigating hybrid methods that utilize the advantages of both methods. A paper made by Nuttapong Chentanez and Matthias Müller investigates creating one such method. They do this by first simulating water on a two-dimensional height field using the shallow water equations. Then they detect when the height field struggles to simulate and generate particles at that point. The particles then later get removed when they’re enter a point on the height field that can simulate again. There are also methods that use particles to simulate the small details on the surface of what. Like splash effects when water collides with another object or wave.

2.3 Erosion Algorithms

(INTRO TO EROSION)

There are multiple different types of erosion in the real world that can be simulated. These include weathering and denudation. Weathering is a small-scale erosion that usually effects stones and rocks. Denudation is an erosion process that mostly results in shaping the surface of an area causing rivers, valleys, or canyons.

Weathering can be split up into two sections. Chemical and thermal. Chemical weathering is the process when moisture makes contact with the surface of a material causing a chemical change. Thermal weathering is the process at which the change in temperature causes material to expand and retracted allowing cracks to form.

Denudation can be split into three sections. Gravity conditioned movement, splash erosion and fluvial erosion. Gravity conditioned movement is when loose material such as soil, gravel or sand moves when it reaches an equilibrium between the gravity and its inner tension the angle at which this happens is called the talus angle. Splash erosion can be described as when raindrops make impact with a material like sand causing an indent in its place. And finally fluvial erosion is the process at which flowing water interacts with the landscape it is flowing down causing material to be remove from the surface being forced down stream with the water.

To simulate these real-world phenomena there are several different approaches that can be taken to acquire realistic results.

The droplet method uses particle to simulate rain drops that collect sediment as they roll down a surface. They then deposit sediment either when they cannot move anymore or they start dropping small piece based on a pre-defined max capacity.

A paper by O.Stava, B.Benes, M,Brisbin and J.Krivanek[16] creates a program that simulates two different real world erosion phenomena. These being fluvial erosion and gravity conditioned movement. They achieve this by using three different algorithms, two for fluvial erosion and one for gravity conditioned movement. To simulate the water, they used the pipe model method that was improved by Xing Mei, Philippe Decaudin and Bao-Gang Hu[15]. The three algorithms they use are force-based erosion, dissolution-based erosion, and material slippage.

The force-based erosion first establishes a sediment transport capacity to determine how much sediment can be carried by each grid cell. This is acquired by getting the tilt angle of the terrain and multiplying it by the sediment capacity constant that determines the strength of the material. And then multiply it by the velocity of the water flow. After this the amount of sediment being carried is then compared to the capacity. If the amount being carried is greater than the capacity, some sediment then gets deposited. After the final sediment amount is calculated for all grid cells the sediment gets transported by the water to a new location. This is done by getting the partial derivative of the velocity multiplied by the amount of sediment being carried.

The dissolution-based erosion is based on the observation that slow moving water penetrates the soil on the rivers floor and creates slow moving regolith that accumulates until it reaches an equilibrium. This is used to create a smooth surface at the floor of a river. This layer remains a liquid as long as the water doesn’t evaporate or the water level decreases. This then leads to the layer to harden and return to soil. This method uses the same equations as the forced based algorithm but swaps out the sediment amount and capacity with a regolith amount and capacity.

To calculate the material slippage the tilt angle for each section of the terrain is calculated. The tilt angle is then compared to the materials talus angle which is the angle at which the equilibrium of the material breaks. This leads to the material to break off and fall to a new location. So, if the tilt angle exceeds the talus angle, then some material will be deposited down the slope.

3 Design / Implementation

3.1 Methods Used

To create the program for this project multiple different methods and algorithms were used to achieve the desired results. Most of the program was made to run on a GPU so that it can run smoothly in real time.

To first generate the terrain a multifractal method was used. To accomplish this a Simplex Noise function which was first developed by Ken Perlin[9] to generate a height map. The Simplex Noise function used in this program was created by (insert ref). The noise function gets repeated by how many octaves have been specified. Each loop gets multiplied by a scaled weight that decreases each octave to achieve an added level of detail. These values all get put together to create the final noise value for each grid position. The multifractal method used in this section was inspired by Benoit Mandelbrot’s theory on fractals[3].

To simulate the water and erosion an algorithm developed by Xing Mei, Philippe Decaudin and Bao-Gang Hu[15] was used. This is a GPU implementation of the Pipe Model Method first developed by James F. O’Brien and Jessica K. Hodgins[14]. For the program to run parallel on a GPU each step of the simulation gets separated into multiple different scripts that are ran one after the other. These steps are, adding water caused by rainfall or a water source, calculating flux output for each grid cell, calculating water movement based on the flux outputs, updating erosion and held sediment, sediment transportation and water evaporation.

The adding water script will have two different choices for how the water can be added. The first one creates random unsorted noise values and places a set amount of water wherever the values are greater than a defined range. This is to simulate random rain droplets. The second choice requires a grid position. It then checks if the current cell is within a defined radius of the position and adds water to anywhere within the zone.

The flux calculation script will be used to update all the flux outputs for each grid cell. Each grid cell has four fluxes being left, right, top, and bottom. These fluxes represent the pipes between grid cells and are used to store the values for how much water is being transferred from the current grid cell to the neighbouring one. The calculation for the left flux would look like

This calculation show the hydrostatic pressure difference between the current grid cell and the grid cell to the left. being the cross-sectional area of the pipe connect the grid cells, being the length of the pipe, being the acceleration due to gravity, and being the height difference between both grid cells. The max function takes the highest number between 0 and the result of the calculation. Then the other three flux are calculated in a similar way. To ensure that the water height does not become negative a scalar value must be calculated. This is accomplished with this calculation

This calculation takes the amount of water in the current cell to calculate a scalar that will be in a range between 1 and 0. The new flux value will then be updated by multiplying the current time step flux by the scalar .

The water movement script is used to calculate the change in water levels for each grid cell and then calculates the waters velocity which is used in the erosion step. The updated water value is calculated using the equation

is the current water height, is the water height of the current step after being updated, and is the change in volume based on the flux of the current time step. is calculated by acquiring all inflow fluxes coming into the current grid cell put together and subtracting it by all the current outflow fluxes put together. After the water height has been updated the velocity is calculated so that its ready for the erosion step. The calculation for the velocity in the x direction would be

is the amount of water that is passing through the current grid cell on the x axis and is the average water height in the current grid cell during the last two time steps which is calculated by adding the current and previous time step water heights together and then dividing them by 2. The amount of water passing through the current grid cell is calculated through

is the grid cell to the left of the current one and is the grid cell to the right. The velocity along the y axis is calculated in a similar way with using top and bottom flux. To ensure no data is out of bounds when calculating, the coordinates are clamped between 0 and the max map size. This ensures that the water or sediment doesn’t fall out of bounds.

The erosion script is used to simulate forced-based erosion. This is accomplished by first finding the sediment transport capacity for each grid cell based on their velocity and tilt angle. The equation for this is

is a sediment capacity constant that can be altered to increase or decrease how much sediment can be carried, is the tilt angle and is the velocity magnitude. After the sediment transport capacity is found it will be compared to the current held sediment for the current grid cell. If the amount held is less than the transport capacity then some sediment will be dissolved from the terrain to the into the water. If the amount held is more than the transport capacity then some sediment will be dropped off onto the terrain.

The sediment transportation script handles the movement of the sediment based on the velocity found in the water movement script. This is done through the semi-Lagrangian advection method which was introduced by J.Stam [18]. This is calculated by taking the velocity of the current cell and using that to take a step backwards to collect data for the current time step. The equation for this is

is the sediment amount dissolved in water and is the velocity along the x and y axis. If the position after the velocity has been added isn’t on a grid space then the closest position is found and used instead.

The water evaporation script is used to maintain a stable level of water through out the simulation. This is accomplished by using an evaporation constant to multiply the current water level in each grid cell by a percentage. This equation looks like

After the water evaporation step has been complete all the data gets passed back through and the process repeats from the start for the next time step.

3.2 Project Structure

To create this project the Unity Engine was utilized for many reasons. One of the primary benefits of using the Unity Engine over creating an engine from scratch is the pre-built components and tools, which allow for ease of use and rapid development. This enables most of the efforts to go into creating the simulation rather than spend multiple hours building up a system to handle the basis to start the project. Unity has good extensibility as it allows a user to easily create custom tools and components that can be applied to any object in a scene. Unity also supports cross-platform development which allows for users to easily use the simulation on different devices. This is useful if the program is used in a game that requires terrain to be generated on launch. When it comes to performance of the simulator, Unity has a range of GPU support with multiple rendering pipelines and GPU acceleration. GPU acceleration is particularly beneficial when it comes to simulating complex computations and large data sets in parallel. Unity uses compute shaders to run complex algorithms. They allow for users to allocate a given number of threads and channels needed to run a script in parallel on the GPU. Unity also has a range of support for multiple graphics application programming interfaces(API). This includes APIs like OpenGL, Direct3D and Vulkan. This is particularly useful for simulations that require high-quality graphics or visual effects.

When it comes to handling and sending data to the GPU having to send multiple large sets of data takes up quite a bit of processing power and with a simulation being updated every frame the performance will take a hit causing the simulation to run a lot slower than it should. So instead of creating multiple data sets that get sent to the GPU this simulation utilizes three render textures instead of using seven large arrays or vectors. Each render texture in Unity has four channels, these being a red, green, blue, and alpha. These four channels are then used as separate data sets. The render textures used are the terrain where the red channel which holds the main terrain body, the green channel that holds the body of water, the blue channel that holds the sediment held at the current time step, and the alpha holds the sediment held at the previous time step. The flux which uses all four channels as for the pipes used in each grid cell for water transfer. And the velocity field which uses two channels being the x and y velocity of the water.

In this project there are two c sharp scripts that are used for handling the compute shader that creates the randomly generated terrain map and the script that controls the entire simulation. These two scripts are both attached to a single object in the scene. The generate map script has no update functions inside it and only gets called by the simulation script when a new terrain map is needed to be generated. The simulation script starts off with initializing all the render textures, mesh and shaders. The render textures are all the same size which is determined by a map size variable. The mesh is generated by creating three arrays that hold the vertices, triangles and texture coordinates. The y coordinate for the vertices are all set to zero as vertex displacement is used at another point to warp the y position into the right place. The x and z coordinates are turned into a percentage, based on their position and then scaled by a chosen scalar. The texture coordinates are the coordinates needed to assign colours or textures in a shader and have to be set between a range of zero and one. These are found similar to the vertices but without using the scalar. The triangles are used to create the faces of the mesh. To create a single triangle requires three vertices to be inputted. The image shown below shows the process of creating the triangles

(image to describe triangles)

The render textures and mesh can be re-initialized at any point to create a new map and reset the simulation.

This object holds two unity made components being a mesh filter and mesh renderer. The mesh filter is used to store the objects mesh data and the mesh renderer is responsible for rendering the object based on the mesh given in the mesh filter. The mesh renderer also requires a material to be added to it. This material is responsible for how the object will be rendered. This can be completed by using one of many of Unity’s pre-built shaders or a user can create a custom shader.

This project uses a custom-made shader. The custom shader used renders the model with a basic lighting technique. This requires a normal map, a colour for the terrain and a light position. The normal is re-generated each frame due to the constantly moving fluids. This is done through a compute shader. The colour of the terrain is generated based on how much water is at the current position. The red colour channel is subtracted by the amount of water and the blue channel has the amount of water added to it. The light position and direction is found by using a Unity library that finds the current light source in the current scene to use as the lighting data. This shader also uses vertex displacement to form the terrain and water into shape. Vertex displacement is used instead of altering the mesh directly because it is performed each frame which will take up a great deal of processing power on the Central Processing Unit(CPU).

--------------------------

Through out the simulation there are multiple variables that can be altered to change the appearance and results of the product. These variables are stored in the two scripts that are attached to the main object in the scene.

The generate map script holds the variables:

Octaves – This controls the amount of times the simplex noise function will be looped for each grid cell.

Lacunarity – This controls how much the scale gets altered by each octave.

Scale – This controls the level of detail used for each octave. This value will increase for each octave which allows for a higher level of detail.

Weight scaler – This controls the amount the noise will affect the end result and will decrease for each octave.

The simulation script holds the variables:

Map size – The amount of pixel size of the render textures and how many vertices there are along the sides of the terrain.

Scale – The size of the entire terrain. The higher the value the larger the distance between vertices there will be. This only affects the x and z coordinates of the terrain.

Elevation – The height of the terrain. This value only affects the y coordinate of the terrain and is updated each frame in the vertex shader.

Animate water – This is a Boolean that controls whether the water is rendered or not. This will allow a user to see the erosion affect the terrain a lot better if it is turned false.

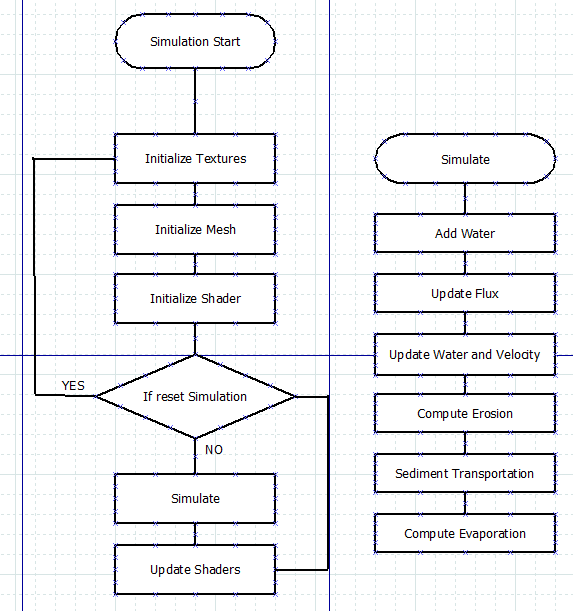
Water increase – The amount of water that is added to the terrain.

Evaporation scale – This controls how much water will be evaporated at the end of each frame.

Sediment capacity – Helps scale the amount of sediment that can be transported by water. It can be increased or deceased depending on how much sediment should be transported.

Dissolving constant – A scalar that controls the amount of sediment that dissolves into the water.

Deposition constant – A scalar that controls the amount of sediment that gets deposited get to the terrain.

The overall flow of the project starts with the simulation script. This starts with initializing all the variables needed for the simulation to run properly. At any point the simulation can be reset at the press of a button. The simulation loop consists of each compute shader getting updated and then ran. The flow would look like this

4 Testing and Results

4.1 Testing methods

4.2 Results

5 Conclusion

References

[1]<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5ca07a56725f8ae6c74778a86a4736ebaab6add6>

[2]<https://onlinelibrary.wiley.com/doi/full/10.1111/cgf.13657?saml_referrer>

[3]<https://users.math.yale.edu/~bbm3/web_pdfs/encyclopediaBritannica.pdf>

[4]<https://epubs.siam.org/doi/abs/10.1137/1010093>

[5] texturing and modelling: a procedural approach book

[6]<https://link.springer.com/chapter/10.1007/978-3-642-84574-1_34>

[7]<https://onlinelibrary.wiley.com/doi/full/10.1111/j.1467-8659.2010.01827.x>

[8]<https://dl.acm.org/doi/epdf/10.1145/325165.325247>

[9]<https://dl.acm.org/doi/epdf/10.1145/566570.566636>

[10]<https://nanomedic.org/understanding-the-navier-stokes-solvers/#:~:text=The%20du%2Fdt%20is%20the,and%20that%20mass%20is%20conserved>

[11] Fluid Simulation for Computer graphics, second edition book

[12]<https://academic.oup.com/mnras/article/181/3/375/988212?login=false>

[13]<https://www.dive-solutions.de/blog/sph-basics>

[14]<https://smartech.gatech.edu/bitstream/handle/1853/3599/94-32.pdf>

[15]<https://xing-mei.github.io/files/erosion.pdf>

[16]<https://cgg.mff.cuni.cz/~jaroslav/papers/2008-sca-erosim/2008-sca-erosiom-fin.pdf>

[17]<https://www.cs.cmu.edu/~112/notes/student-tp-guides/Terrain.pdf>

[18] <https://pages.cs.wisc.edu/~chaol/data/cs777/stam-stable_fluids.pdf>