Dissertation (draft)

# Abstract

(General summary- explain purpose of application, potential real-world applications, limitations of existing simulation models)

# Contents

# Introduction

This portfolio project will investigate the methodology of creating rivers in videogames and create a tool to assist with the creation of digital landscapes. Digital representations of geographical features are often created by artists, with little or no reference to real-world geographical data. While artistically impressive landscapes are prevalent in modern videogames, especially those in an open-world environments, the geographical basis behind these features is often forgotten, producing inaccurate or unrealistic rivers and pools. The rise of procedurally generated games also often turns a blind eye to real-world data, instead opting for a simple representation of “a line of water” (https://ieeexplore.ieee.org/abstract/document/7295776). A tool that could generate a landscape, simulate years of fluid movement and erosion, and then provide accurate soil and landscape height data to artists could assist in ensuring artistic landscapes use a more realistic basis.

Node-based simulations of sediment pick up and deposit could allow a digital representation of hydrological landscape features. Such a system would allow for the creation of complex geographical features found in rivers (such as ox-bow lakes) that are scarcely seen in artistic landscapes and assist in the creation of far more naturally-inspired rivers. Features like bank erosion and sediment transfer(https://www.therrc.co.uk/MOT/References/EA\_DEFRA\_Sediment\_transport\_and\_alluvial\_resistance\_in\_rivers.pdf) are rarely present in these representations due to their complexity(http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.449.5576&rep=rep1&type=pdf) and computational intensity to simulate(<https://huw-man.github.io/Interactive-Erosion-Simulator-on-GPU/>).

Particle-based systems can also be used to streamline fluid simulation due to their more simplistic mechanical nature- it may be impractical for an assisting program to run on a landscape for several hours if a map is constantly being iterated on by artists. Various fluid simulation methods exist using node-based, particle-based, and mesh-based systems, and analysing various options and finding the correct simulation method will be vital in the creation of such a tool.

Aims

The aim of this project is to create a tool that generates a randomized 3D landscape and manipulates it to form rivers and lakes through simulation of fluid dynamics, exploring the effects of sediment acquisition and deposition. The tool will create a landscape with natural-looking height differences, a simple representation of foliage, and soil maps determining the exact properties of the land at any given point. It will be able to run a rainfall and spring water simulation, showing the effect that fluid would have on the landscape through means of erosion and sediment deposit over multiple years. Fluids should be able to flow through the landscape and form pools, causing erosion and behaving as they would in traditional fluid dynamics (http://www.jlakes.org/config/hpkx/news\_category/2015-06-01/PhysicsofLakesVolume3MethodsofUnderstandingLakesasComponents.pdf) , transferring solids from banks and cliffsides. Simulation will be particle-based, representing the movement of fluid down the landscape, and forming pools as it comes to rest. The precision of the model should be variable, allowing for both large and small-scale simulation, from a single riverbank to a kilometre squared of land.

The simulation will not account for the effect of humans or animals on the landscape, the freezing and thawing of water, or the effects of altitude on the behaviour of fluids and gasses. The inherent complexity of these features would require a full study (https://www.sciencedirect.com/science/article/pii/S2211379717302437.) and would prove difficult to simulate in the scope of this project.

The key objectives of this project are:

* Determine the optimal method of fluid simulation for use in creating a map creation tool
* To create an algorithm to generate a random landscape with varying terrain height, existing bodies of water, and a large range of soil and rock types.
* To simulate the formation of rivers and flow of water over the generated landscape, including the effect of rain and spring water flowing and pooling without use of a pre-defined river spline.
* To simulate the erosion of terrain and deposit of sediment through fluid dynamics and force calculation to manipulate the generated landscape over time.
* To allow specification of map properties, so the user can change both terrain generation and water behaviour to affect the simulation outcome. Such examples are hill height and rarity, chance of natural spring generation, scale and size of landscape generation, and the amount of rainfall to simulate within a given year.
* To compare the results of this updated model with real-life geographical data and iterate for maximum realism.

# Literature Review

Digital artistic representations of fluid behaviour are most commonly seen in videogame landscapes. It can be argued that even the most basic representation of fluid movement, such as the logs in Frogger (1981) (citation), are an artistic take on the way water behaviour can affect a game. However, in the modern era, complex open-world games such as Red Dead Redemption 2 (date) (citation) provide artistic representations of real-world locations such as the Hudson River. These representations often stem from artistic works referencing the same area, such as those of Albert Bierstadt (<https://www.polygon.com/red-dead-redemption/2018/10/26/18024982/red-dead-redemption-2-art-inspiration-landscape-paintings>)

Geographical experts are sometimes involved in the development of these digital landscapes, due to the complex nature of fluid behaviour and variety of landscape features that may form as a result (https://core.ac.uk/download/pdf/301635894.pdf). Many games, however, neglect reference to such behaviour, in favour of performance instead. Games such as Rimworld(date)(citation) use randomly generated spline-based river generation, which can often cause unintended behaviour and unrealistic-looking landscapes.

Landscape splines are often used in many non-procedural games too(https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Landscape/Editing/Splines/) , designating the points at which a river flows to, and automatically filling the area with a simple fluid model or plane. While this can provide an accurate representation of man-made channels, it often fails to look create natural-looking areas of water. Ideally, fluid simulation could be used alongside or instead of splines, to create realistic landscapes with the correct soils and sands surrounding bodies of water.

Landscape generation is a well-studied field, and various techniques have been explored in the past. Scenery generation tools are frequently used in both videogame and movie production (<https://en.wikipedia.org/wiki/Scenery_generator>), having been used on both the King Fu Panda and Pirates of the Caribbean movie franchises (citation). These are often generated using Perlin or Simplex noise (citation), relying on manual modification by artists to produce realistic weathering or foliage effects. To generate a simple yet realistic landscape, prior studies suggest that modified Perlin noise to generate a heightmap provides a good foundation to create geometry (https://core.ac.uk/download/pdf/250147208.pdf).

The program was initially planned to use a variation of the Diamond-Square algorithm for procedural terrain generation (https://en.wikipedia.org/wiki/Diamond-square\_algorithm), however the lack of flexibility and control over what terrain is generated required a different approach. In order to allow full control over the terrain generation parameters, a different approach was required. Fractal landscaping (<https://en.wikipedia.org/wiki/Fractal_landscape>) was also an option, but the lack of a defined square grid and lack of control over exact map specifications would make particle fluid simulation difficult. A custom-made generation tool was the only solution to truly fulfil the application’s aims.

Studies into fluid simulation for games have been used to generate pools and simple moving bodies of water on large-scale environments in the past (<https://ep.liu.se/ecp/034/010/ecp083410.pdf>), but it is currently not a commonly-applied practice. In theory, this could be simplified using authored or generated landscapes, then running a fluid simulation representation to ensure that it is geographically accurate. Fluid simulation representations have existed for centuries, since the time of Archimedes(citation), but computational methods allow for such representations to be run on a far larger scale. In game world creation, it could be used to provide geographical representations of real pedological features.

Many studies into fluid simulation exist, with varying scopes, scales, and approaches. Multiple algorithms exist to simulate fluid movement, with varying usefulness in tackling this project. One such example are the Euler Fluid Equations <https://levelup.gitconnected.com/create-your-own-finite-volume-fluid-simulation-with-python-8f9eab0b8305>, which emphasize having an incompressible fluid of constant density within a closed system. This is calculated as a flow velocity vector for points on a grid, considering all body acceleration and acting forces, such as gravity. These would prove perfect for this project, being both easy to simulate and considering the surrounding environment, as well as utilizing a similar node-based system. However, it is possible to hit a point of singularity, a possibility that is very likely to occur in a large-scale simulation due to the number of calculations being performed. <https://cmsa.fas.harvard.edu/euler-workshop/> .

Another example are Navier-Stokes equations <https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations> . These serve as an alternative to Euler Fluids, focussing on the conservation of mass and momentum at given points within a liquid’s surface, in a similar node-and-vector system. These can also account for temperature and viscosity, as well as both compressible and incompressible flow, allowing more complex simulation of fluids in multiple states. A common problem with Navier-Stokes equations is the fact they are infinitely differentiable (citation)- for any given point in the fluid’s domain, the vector velocity of the point can be infinitely refined- no answer will be 100% accurate. While this doesn’t cause a huge issue in terms of large-scale simulation (there will always need to be refinements, as true simulation on a particle-level would take an infinite or near-infinite amount of time [https://en.wikipedia.org/wiki/Quantum\_computing](https://en.wikipedia.org/wiki/Quantum_computing#:~:text=Quantum%20computing%20is%20a%20type,are%20known%20as%20quantum%20computers)) a cut-off point would have to be found in order to accept a solution with an acceptable level of accuracy. The Cauchy stress tenor <https://en.wikipedia.org/wiki/Cauchy_stress_tensor> of a unit space can be calculated to assist in the solving of these equations, but accuracy is still limited within a reasonable timeframe.

The Lattice-Boltzman algorithms <https://en.wikipedia.org/wiki/Lattice_Boltzmann_method> avoid solving these equations by simulating a fluid as a lattice, with tension and relaxation points. The algorithm is very adjustable, mimicking both vapours and fluids on small scales(citation). However, complex boundaries significantly complicate the algorithm, and it operates better for small-scale fluid simulation, such as deformation of a single droplet (citation).

Initially, the aim was to implement an algorithm to solve the Navier-Stokes equations to an acceptable degree of accuracy, as other fluid simulations have used in the past (<https://ep.liu.se/ecp/034/010/ecp083410.pdf>). However, when paired with the larger-scale of the environments of the planned simulation area (estimating one km^2), preliminary testing revealed that the performance would be unacceptable. The solutions would be either far too inaccurate or take such a significant amount of time that the program would be impractical, potentially running for several minutes to simulate a single year of fluid movement. While acceptable for a flooding-avoidance program or smaller-scale animation render, this would be impractical for my program’s purposes, which should allow landscape deformation in a reasonable amount of time (under an hour.)

A study by Nicholas McDonald into the movement of water through a simulated particle on a grid, called the “Hydraulic Erosion Algorithm”, proved a better basis for this study(citation). Using a previous study of sedimentation and mass transfer, a simple demonstration of water moving as a particle to form a river was developed, allowing streams to form in the terrain (<https://nickmcd.me/2020/04/10/simple-particle-based-hydraulic-erosion>.) Although more rooted in classical mechanics than traditional fluid simulation, it could provide an acceptably accurate representation of water moving through a landscape, while keeping computational time reasonable. It also provided a basis for a realistic representation of sediment pick up and deposit, a key focus in this project. McDonald’s pooling system, while effective on a smaller scale, proved to be computationally intensive on a large scale, so a custom flooding method was developed for this project.

Alongside the Hydraulic Erosion Algorithm, the program aims to implement additional handling for a soil map for the landscape. Instead of treating a heightmap as a deformed plane, it will use information about the terrain type and underground structure of the landscape, allowing for “true” erosion in which rocks and differing kinds of soil can be unearthed, as well as deposits developing on the edge of rivers. This would allow for a far more accurate representation of a river bank, which could be used as artistic reference during map creation for a game. Soil maps are often used in real life, when taking samples of farmland (citation) or geographical surveys (citation), allowing easy comparison of simulation results to those in real scenarios, using publicly-available data such as the SSURGO dataset (citation).

The Hydraulic Erosion algorithm also considers the behaviour of foliage on the landscape, simulating tree spread and growth. However, McDonald’s study was on a smaller scale than what is planned to simulate, and the computational cost of simulating individual trees independent to the map’s node-based system is too high. The program will instead use a less dynamic method of vegetation representation, using a foliage coverage percentage per meter squared.

# Methodology

The Map class is the access point of the simulation- it is used to house all the geographical and hydrological data of each square meter in a two-dimensional array of data structures called Nodes. Each of these Nodes houses the individual soil map for the given area, any particle and foliage levels, and pooling data for fluid simulation. They can function as a 2D height map or mesh coordinate values for a renderer, storing colour and height values for each layer of the soil map in NodeMarker structs. These are frequently manipulated by Plant and Drop classes that are created during the simulation runtime. The fluid simulation is all performed in the Drop class, directly modifying map nodes that surround simulated particles with amounts of sediment pick up and deposit. Functions in the Plant class are then used to modify foliage values on affected nodes, destroying and creating plant life dependant on water movement. The map also stores known definitions of soils, allowing easy comparison and determination of soil types at runtime.

A MapRenderer class houses the accessing of data for visual representation, as well as an OpenGL wrapper that visually displays results to the user. This uses several OpenGL Shaders, stored as ShaderProgram classes. Interaction with the program uses the SDL event system and usage of the console, to access individual coordinate data.

Diagram

Description automatically generated with medium confidence

Terrain Generation

Generation of terrain is performed in the constructor of the map class- this will define the soil types used within the program, populate the nodes of the map with heightmap data generated using Perlin noise (citation), and generate variations in soil type and foliage.

Generation of a terrain heightmap uses various properties defined in a MapParams struct, provided in the map constructor. MapParams contain all tweakable values that can be used by the program, controlling rarities for terrain types, the scale of the map, rates of change for soil and rock types and foliage spread chances (among many more options.) This allows the program user to generate a terrain matching their specifications- for example, a 10km\*10km terrain with steep inclines could be generated by decreasing the hillRarity, increasing the hillHeight and increasing the scale. Eight instances of Perlin noise are generated and used in map creation, all generated from a defined seed (or randomly seeded if no value is given.) A singular seed will always generate the same noise values for a map but changing the parameters will alter the effect is has on terrain. For example, to expand on a map that is already generated, the scale can be increased to see what would be beyond the map borders, at a lower precision level.

Chart

Description automatically generated

*(Usage of scale increase on a consistent map seed. Note the loss of detail on the right-hand extension of the pool due to the decreased precision)*

Perlin noise, modified using map parameters, is used to define the following for every point on the grid:

* Base Variance- the inherent inconsistencies in terrain. These represent terrain modification from roots, animals, and any other minor terrain-altering affects.
* Hills- large peaks and troughs of terrain, defining the general structure of the landscape. The frequency of hills defines the general steepness of any given point on the terrain.
* Divots- smaller areas of raised or lowered land (by default, around 10m^2 area.) Often the forming points for natural pools or shaping a rougher hill or mountainside.
* Mountain- huge raises in terrain, often peaking in a plateau or point. Very rarely occur but make significant effect on the terrain when they are generated. Mountains can be generated more frequently by changing the MapParams.
* Lie- the height of the terrain compared to sea level. This will affect the change of the generated map being waterlogged, or completely dry.
* Rock- the frequency and density of generated rocks within the earth. Higher rock frequencies can cause huge cliffsides to be generated, or massive boulders to be unearthed by fluid movement.
* Resistivity- The general properties of the soil. Although simplified to resistivity in name, this noise is used to define exactly which types of dirt fall in which areas of the map. High rates of change can be used for sedimentary, compounded soils, whereas low rates of change can be used for a consistent area with little variation.
* Sand- The point at which sand is generated on the landscape. Defaulting to appear near water, sand is the only assumption of pre-performed hydrological erosion used by the terrain generation. Higher sand intensities will provide loose, sandy soils which are often low in fertility, alongside massive sandy banks, and beaches on coastlines.

The heightmap of the terrain is populated using the base variance, hills, divots, mountains, and lie of the terrain, before being filled to a default water level specified by the parameters. Using this heightmap, the values under the topsoil can be populated. Each node consists of various NodeMarker structs, which define the soil properties at that given level. These are populated in the addRocksAndDirt function, which iterates through the terrain, populating significant changes in soil or rock properties. An example node may contain this data: A picture containing diagram

Description automatically generated

*An example of populated node data*

The populated soil map can also be viewed using debug functionality, by rendering the map at a specific height to segment it and see soil data at any given level. Any erosive functionality prioritizes the top layers of soil, however in cases of large amounts of sediment being gathered, it is possible to completely eliminate some layers and erode at lower levels. In the example above, a significant volume of water in a single year could remove the whole topsoil layer and erode at the higher resistivity soil underneath. High erosion levels for several years could reveal the rock layers under the soil. Resistive forces from the soils and rocks are considered when eroding, so this is likely to happen during any given simulation.

Bedrock does occur higher in map generation than it does in real-life data, as well as being flat as opposed to variable, and this largely due to performance reasons. Except in extreme cases, it is very unlikely that any point on the map will reach bedrock level. If it does, the lie parmeters of the land can be increased so that more containing rocks and soils are generated below the surface.

A picture containing text

Description automatically generated

*An example of segmented terrain, viewing the soil and rock underneath.*

The values of soil can also be analysed using the getSoilType function. This iterates through all known soil types for the map and finds the best match at the given node. If a loam (top layer of deposit that is different to the bulk of soil for that node) is present, it will also be displayed. Text

Description automatically generated

*An example of the getSoilType function*

The default values for soil parameters in the program, and their randomization range, are based on the SSURGO dataset (citation) which contains soil data for the mainland of the United States of America. The most common soil type, eapa (citation) is prevailant throughout the majority of generated terrain before any hydrology simulation takes place. As sediment deposit takes place, a ethridge laum is likely to develop, which is often seen on real-world riverbanks. However, the actual values of the program can be altered using the map parameters. A high-clay area will have soils that behave completely differently to a high-sand concentration, and experimentation is key to determine the optimal lanscape layout. Debug functionality is also available to erode all terrain simultaneously, or to strip the top NodeMarker from every node, to experiment with these values’ outputs.

OpenGL visuals

All terrain rendered by the application consists of tiles, each of which represents one node’s data. A tile is a square made up of four smaller triangles, each of which has its positions modified by the GLSL vertex shader to manipulate it into the correct position. These positions are supplied by the map’s node data, giving height, colour, and water depth values for both the current and surrounding nodes. Knowing the surrounding node values is important, as the grid system should not be obvious in the visual representation of the terrain, so all colour values must be linearly interpolated so individual tiles do not stand out. This interpolation is performed in the shader, as it is performed once per tile per render call and processing it on the GPU allows for faster rendering.

An additional method for performant rendering is LOD scaling (https://docs.unity3d.com/Manual/LevelOfDetail.html). During a render call, the current zoom level of the map and the camera position is considered when processing node data. If the map is sufficiently zoomed out so that rendering every tile could cause performance issues, some node data is skipped, rendering 16m^2 tiles instead of a 1m^2 one. The bilinear interpolation in the shader also interpolates between only the corner nodes of each tile, reducing the workload on the GPU. This allows for incredibly large landscapes to be rendered at very low zoom levels without rendering up to 1,000,000 tiles to represent 1km^2 of land.

A picture containing shape

Description automatically generated

*An example of LOD scaling on map tiles*

Another method of improving performance at closer zoom levels is culling- tiles behind the camera, and tiles a significant distance away, are not rendered when closely zoomed in. This scales inversely to the level of detail, as skipping node data can allow for a larger distance to be rendered.

All pool visualization is performed in the fragment shader. Still pools of water can be represented by a given water level for the tile. For each fragment within the tile, the height of the fragment is compared to the water level at that point in the tile, obtained by using bilinear interpolation between neighbouring points(https://en.wikipedia.org/wiki/Bilinear\_interpolation). If underwater, the tile’s colour is modified to be tinted blue, creating a simple boundary effect. A fully submerged tile is all tinted blue, whereas tiles on the water-land boundary can be partially submerged in fluid. This creates a simple yet effective method of displaying seamless tiled water.

A picture containing nature, dark, clouds

Description automatically generated

*An example of water at a boundary. Note the tile borders are indistingushable.*

A depth map was used in earlier development of the program, to represent water depth in a similar way to the area in which light would hit on a shadow map (citation). However, common issues with shadow mapping such as acne (citation) and peter-panning (citation) had increased visibility due to the complex nature of the terrain and the precision needed along the water shoreline. 

*Depth map water rendering, with acne occurring on the land’s edge*

Particles and foliage are both represented by changing the colour of the tile as it is supplied to the shaders. This behaves similarly to a variation in terrain colour from soil deposit, but instead of the node’s soil data, it is based on the foliage and fluid data of the node. A potential expansion point in future would be a more visually impressive modification of these methods, as while visual fidelity is not the focus of this project, the tinted values can sometimes be harder to see than the more colour defined pools. Individual tile colour is also modified by terrain height, to allow the user to differenciate between height levels. This is strictly a capped interpolation between the lowest and heighest points on the map (within reasonable bounds) to allow glance-value comparison of areas without having to examine any node data.

To simulate rainfall, volumes of water are spawned at random points of the landscape, and descend and cascade to lower points, picking up and depositing sediment as they move. Particle movement over the terrain uses classical mechanics to calculate changes in velocity and force exerted on the landscape. The descend function is repeatedly called while a particle still contains a significant volume of water. This function uses previous water movement, existing velocity, frictional forces from terrain and foliage, and the gradient of the current tile’s slope to calculate exactly where a simulated particle should move.

Initially, several checks are made to ensure the particle is not behaving in any unrealistic manner. If it has little or no volume, has moved off the map, or is trying to descend through a body of water, it is immediately terminated, and simulation for that particle ends. Any carried sediment is immediately deposited, and it attempts to disperse itself.

If the particle is capable of movement, it checks surrounding nodes for any existing water flow. If possible, it will join an existing stream of water, obeying the flow of other water particles in the same area. Frictional forces are then applied based on the particle’s initial velocity using mechanical frictional formulae where m is the particle mass, g is the gravitational constant, μ is the frictional coefficient (determined using the foliage coverage), n is the normal force of the surface and Θ is the terrain normal angle.

Text

Description automatically generated

*Mechanical friction calculation*

The force of gravity is then applied to the particle, accelerating it down any slope that it may be travelling on. The calculation and parameters are similar to that of the frictional calculation, as they are both calculated from the normal force.

Text, letter

Description automatically generated

*Acceleration from normal force*

These calculated acceleration values are for a fixed timescale of 1 second. These need to be multiplied and scaled to apply to the extended period of time at which it is assumed the particle is descending for an accurate representation, to ensure it visits every node on its path.

Once the particle has descended, it cascades. This is a representation of the pickup and deposit of sediment around the area of particle movement due to changes in velocity. Van Rijn’s suspended load transport formula (https://www.leovanrijn-sediment.com/papers/Formulaesandtransport.pdf ) can be used to calculate the average pick up and deposit of sediment in any given particle movement, where qs is suspended-load transport (pick up of sediment in kg/m2/s), αs is the acquisition coefficient (0.012), ρs is the relative density of the solid (a derivative of resistive force in this model), U is the force required for the current change in velocity of the given particle, d50 is the particle size (assumed at 30000 microns, average for dirt & sand), g is the gravitational constant, and D\* is the dimensionless particle size (once again, assumed at 30000 microns)

Logo

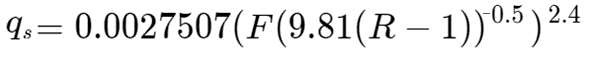
Description automatically generated with medium confidence

A picture containing text

Description automatically generated

*Van Rijn’s suspended-load transport formula (with velocity simplification)*

Using an assumed standard particle size and substituting in values for constant particle sizes, the acquisition coefficient, particle density, and gravitational constant allows this to be reduced to :



Where F is the acting force on the area (calculated using F=ma for the velocity change), and R is the resistive force of the current ground layer (specified by soil/rock type.) This allows an accurate determination of the amount of sediment picked up through any given particle movement on the grid. Other methods do exist for calculating sediment pick up, including wave and cliff erosive force calculations (<https://www.journals.uchicago.edu/doi/abs/10.1086/628340>) however due to the lack of wave simulation and over-simplification of these parameters, Van Rijn’s studies proved to be a better basis for this simulation.

Any picked up sediment from the node is added to the particle, mixed with whatever sediment was already carried. This is stored as a weighted average of all sediment in the particle, representing a mix of the individual solids stored within. For example, a particle carrying both stone and soil erosive material would be deposited as stony soil, which is both infertile and lower in clay content than usual topsoil. This material is created by weighting the properties of both the soil and stone build ups- the soil will have higher fertility, and the stone a lower fertility. Higher concentrations of stone compared to soil will result in a lower fertility sediment, while high soil concentrations with smaller stone numbers will result in a fertility just below that of the starting soil. This can be calculated using the below equation, where *φ* is the new fertility of the sediment, *φ1* is the fertility of the material being picked up, *φ2* is the fertility of the existing material in the sediment, m1 is the mass of the material being picked up, and m2 is the mass of the existing sediment in the particle.

A picture containing text, clock, watch

Description automatically generated

Diagram, box and whisker chart

Description automatically generated

This method is replicated for all properties of the eroded material, such as sand and soil content. The combined material is then ready to be deposited on the landscape as a loam on another node. The amount deposited can be calculated using Stokes’ law (https://en.wikipedia.org/wiki/Stokes%27\_law) Stokes’ law states that the drag force acting on a particle moving through a fluid can be determined by the following formula, where Fd is drag force, η is fluid viscosity, r is the radius of the particle, and u is the terminal velocity of the particle.

Text

Description automatically generated with low confidence

At terminal velocity, the drag force acting on the particle is equal to the force of gravity. The force of gravity acting on the particle can be determined using Lamb’s calculation of settling velocity (<https://www.abebooks.co.uk/Hydrodynamics-Dover-Books-Physics-Sir-Horace/31164960207/bd?cm_mmc=ggl-_-UK_Shopp_Textbookstandard-_-product_id=UK9780486602561USED-_-keyword=&gclid=Cj0KCQjwpcOTBhCZARIsAEAYLuVH7JO9k55_QXkMXO4-bbA3cNkuqpyzvvJzxuuCBqQplhvz6AECs5MaAr__EALw_wcB>) where Fg is the acting force of gravity on the object, ρs is the density of the solid, ρf is the density of the fluid, g is the gravitational constant, and R is the radius of the spherical particle.

Text, letter

Description automatically generated

Equating both forces as Fg =Fd and solving for u presents the following formula, for particle terminal velocity can be calculated. (<https://pdf4pro.com/amp/view/stokes-law-settling-velocity-deposition-37ebeb.html>) u = terminal velocity of a single particle, d is the diameter of a single particle, g is the gravitational constant, ρs is the density of the solid, ρf is the density of the fluid, and η is the viscosity of the fluid.

A picture containing text

Description automatically generated

Assuming our particle width and assuming our fluid is rain water (as it will be during the simulation) this can be rearranged to calculate the fall time for any particle at terminal velocity. This can, in turn be multiplied by a time constant for the simulation tick to calculate the amount of sediment that would settle at that point in the simulation. Therefore, the amount of sediment deposited in a linear flow can be calculated as a multiple of the amount of sediment stored in the particle, where l is the amount of sediment deposited, m is the mass of sediment stored in the particle, ρ is the density of the solid, and u is the particle speed.

Text

Description automatically generated

This sediment is placed at the surrounding nodes to the particle position with every descent, placing small amounts of sediment build up as the fluid travels and creating river banks alongside frequently-travelled streams. All visited nodes are added to a “track” array, increasing their chance of usage by future particles.

When a particle has stopped moving for any reason, or has entered another pool, it floods. Flooding is a particle either forming or increasing the volume of a pool of water at its location. A “plane” value is calculated, which is equal to the height of the node the particle is stopped at, plus an additional increase height. A flood fill (<https://en.wikipedia.org/wiki/Flood_fill>) is then performed at the plane value, using a stack and list of accessed nodes for storage, encapsulating the whole area that could potentially be filled with water. This is then compared to the volume of the particle. If the particle can successfully fill the area, all nodes within the set are filled to the plane’s height value. If the particle’s volume is incapable of filling the area, then the algorithm is repeated, decreasing the increase height. If the particle’s volume is capable of filling the area, the algorithm is repeated until the volume of the particle is fully used, decreasing the increase height if needed. If the increase height is sufficiently low at the end of a cycle, several outcomes may occur.

A picture containing diagram

Description automatically generated

*A two-dimensional representation of the filling algorithm*

* If the particle has created a new pool and used the majority of its volume, it is accepted as a “good enough” fill, and the excess volume is lost to evaporation.
* If the particle is trying to fill off the map, it is assumed that it is unable and it is immediately lost to evaporation.
* If the particle has failed to create a new pool and cannot increase the current water height even a slight amount (such as if it expired on a large cliff and tried to fill the whole landscape) it is assumed to be a trickle of water leading to a location at which it evaporates, so is ignored, and treated as a stream.
* If the particle has entered a larger pool of water and used its volume to increase the pool, it is accepted as part of the pool and its volume is lost.
* If the particle has entered a larger pool of water and failed to increase the pool volume, an overflow occurs.

An overflow is a particle exiting a pool from a drainage point, as the pool’s value can no longer be increased. The pool will not stagnate, as it is receiving a steady flow of water, but due to the limits of its supply, it will never increase in size significantly. Any water entering the pool will simply be assumed to drain out the lowest border of the pool as a particle. The border is slightly eroded through the steady stream of water exiting it, which can decrease pool capacity for future simulation ticks. The whole pool will be filled to the drainage point as soon as an overflow occurs.

Shape, polygon

Description automatically generated

*A two-dimensional representation of an overflow*

Should the particle immediately re-enter the pool, it is assumed to be lost to evaporation, as it would enter a loop until its volume was fully decreased. Combining both the fill and overflow algorithms can allow for large hydrological structures to develop, such as multiple pools down a mountainside, all overflowing and feeding into the next. The combined particle-and-pool based structure allows these to behave as they would in real life, providing insight into how fluid might behave on the generated landscape.

Foliage & plant representation in the application is significantly simpler than fluid simulation, as it is not the focus of the program. Similar applications often use a method of tracking individual plant positions on smaller areas for visual fidelity, as individual variations on plant matter can be rendered at each position. For simplicity and performance reasons, this proved unfeasible on larger landscapes. Testing with individual plant tracking meant that thousands of entities were being added to a vector at runtime, which slowed simulation times significantly for very little benefit, often taking up to 10 seconds. Due to this, a foliage coverage value for each node was used instead.

The grow function, called once per simulation tick (representing a calendar year), serves as the basis for modification of foliage coverage. Firstly, long-range fertilization can place plants randomly on the landscape, as a representation of long-distance seed transfer from birds or gusts of wind (<https://besjournals.onlinelibrary.wiley.com/doi/10.1111/1365-2745.12690>). Then, operations are performed on foliage in every node on the map.

* If the node is void of foliage, nothing needs to be performed.
* If the foliage is sufficiently dense, it has a chance spread to neighbouring nodes, spreading across the map.
* If the foliage is in a pool or stream of sufficient depth, it is immediately killed, as it cannot grow underwater. If the foliage is in a stream of very little depth, it can continue growing. Foliage will grow faster around the borders of fluid.
* Foliage may randomly die off in areas, representing possible blight, wildfire, or effects of animal behaviour.

Spreading foliage may fail if the correct conditions aren’t met. These could involve overpopulation, if surrounding density is too high, water presence, or being on an incredibly steep slope. While simple, this provides a rough representation of how foliage might behave on the given landscape and allowing for all values to be changed by the user (for example, the chance to spread can be modified, or the restrictions on overpopulation or ground slopes can be removed.)

# Critical Reflection

Due to the investigative nature of this portfolio project, several changes were made to methodology as research into the simulation topic furthered. While a fluid simulation approach that strayed away from McDonald’s particle model was initially considered, the solving of Navier-Stokes equations combined with sediment calculations, which could not be performed as a particle due to the use of vectors instead of particles in the Navier-Stokes models, proved both computationally intensive and difficult to accurately simulate. Heavy assumptions about particle behaviour and sediment transfer would have to be made in order to get the program to run successfully. While performance is not the true focus of the project, it is important for the simulation to be performant enough to not inconvenience the user, who may want to re-run simulation with varied parameters.

Significant changes to rendering technology were adopted- the proposal for this project theorized that treating the terrain as a single mesh would allow for nodes to be represented as a deformed plane, but this proved both difficult to create and to debug. Due to the closed nature of OpenGL rendering, external tools such as RenderDoc (reference) must be used to see individual draw calls, and the nature of a single mesh meant that debugging individual elements was nearly impossible. The tiling method used in the final program is slightly less performant than modifying positions on a single mesh, for which the LOD system was developed to assist with, but proved far easier to debug and did not require an additional texture render to colour individual tiles. The representation of fluid pools on a larger mesh would also prove difficult, and the shadow mapping-inspired method (see section X.X) was developed as a solution to this. Due the problems using a depth map for fluid rendering and the lack of accessible debug data, the tiled method has proved a greater success.

The map parameters and NodeMarker system as a representation of the soil map proved to be two theories that were initially planned and performed as expected. The soil map is a significant expansion on McDonald’s hydraulic erosion model, allowing for much more precise transfer and sampling of soil data.

While McDonald’s rough design was used for the descend/fill system of fluid representation, improvements were made to the model to allow for more accurate simulation. Calculation of real-world forces and friction during particle descent and an overhaul to the pool filling system allow for much more realistic results, as well as increased user control over fluid simulation, as is suitable for a tool to assist in map creation. McDonald’s proposed flood method (with drains calculated for any given flood) proved incredibly taxing on CPU performance, taking an unreasonable amount of time to run on larger landscapes. Using larger maps, it often led to stack overflows, as the incredibly recursive fill function filled the stack quickly. The modified flooding system allows for quicker, and more accurate, area flooding, without concern over hitting stack limits.

Sedimentation is also significantly improved- the Hydraulic Erosion model uses a given mass transfer coefficient to modify terrain height, the produced application uses Van Rijn models of pickup and Lamb’s equations to calculate sediment acquisition and deposition rates, allowing for behaviour far more similar to real-life solid transfer rates. The exploration into the scientific theory of soil transfer and exact particle behaviours when suspended in fluid is an expansive topic to explore, and the application opts for a far more scientifically-based approach than an acquisition constant.

Initially, an additional focus of this study was to compare the performance impact of simulating fluid movement on the CPU and the GPU. Previous studies have shown the GPU simulation can prove to be significantly better at handling high-load simulation data, and it was planned to investigate the significance of changing the processor that the program ran on. Due to project time constraints and a deeper focus on the comparison of simulation data with real-world examples, these comparisons were cut, and the project was instead developed for CPU simulation only. Were additional time available, comparing the simulation efficiency on the CPU and GPU would prove an interesting focus, and it’s likely that outside of this project’s constraints the majority of simulation calculations would be ported to shaders, allowing them to be run on the GPU instead.

Conversion of mathematical theory into code is sometimes difficult to replicate, and this project was no exception. Assumptions such as particle size, density’s relation to resistive force of material, and the behaviour of fluid as a particle rather than an incompressible mass all limit the accuracy of the finished application. While dynamically changing all these behaviours would provide optimally accurate simulation data, variables with incredibly small variance and many parameters (such as Van Rijn’s acquisition coefficient, which is assumed at 0.012 for the program but can actually vary between 0.008 and 0.0012 depending on many particle properties and transport rates)(https://www.leovanrijn-sediment.com/papers/Formulaesandtransport.pdf) must be estimated. Calculating individual values for each of these assumed properties would have negligible effects on the program’s data but can significantly increase the number of mathematical calculations per particle simulation.

Results of final implementation- look at the simulation in depth and probe for any particular strengths or weak points. Examples of landscapes with varying properties and the effects of water on them.

Lots of diagrams, examples, code snippets, etc.

What have I accomplished? Compare with existing models & simulations in terms of realism & representation. Looking back, would I use a node-based or particle-based simulation?

Did I hit my targets? Can I simulate an ox-bow lake?

# Conclusions

Improvements/time constraints. What could I add? (Temperature, more varying sediment types, complex foliage, etc.)

Analysis of results and **comparison with real life data.** This will require additional research into real events & geographical features.

# References & Appendices