Dissertation (draft)

# Abstract

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

# Contents

Introduction

Aims

Objectives

Literature Review

Artistic Representation of Landscapes

Landscape Generation

Fluid Simulation Methods

The Hydraulic Erosion Algorithm

# 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 with a 1km\*1km map, it will be called 1,000,000 times per render call.

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 edge pieces can have areas that stick out the water. 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.

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.

Visualisation of fluid behaviour- particles & foliage

LOD

Drop & fluid as a particle

Descend

Cascade

Flood

Foliage & plant representation

Potential advantages/disadvantages of these approaches compared to traditional methods

Equations used for fluid dynamics- this is a focus as I’ve been modifying these for the program’s behaviour. Show working and approach to these problems.

# Critical Reflection

How I actually implemented the program. Explain development cycle, any potential issues (discuss mesh deformation tech and plan changes there)

Talk about any scope changes (cut back on the CPU vs GPU focus and more on simulation?)

And tech/mathematical changes to methodology?

Explain final implementation and how it came to be. Screenshots of program running, potential code snippets of mathematical implementation in C++. How to convert from series of equations into readable code?

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