Building Dynamic & Interactive Natural Game Worlds

by

Jeremiah Dunn, BSci

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2015

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Jeremiah Dunn

August 29, 2015

Permission to Lend and/or Copy

Acknowledgments

I would like to thank Lucy Bofin for her support during the duration of my masters, my family and my supervisor, Mads Haahr.

Jeremiah Dunn

University of Dublin, Trinity College
August 2015

Building Dynamic & Interactive Natural Game Worlds

Jeremiah Dunn
University of Dublin, Trinity College, 2015

Supervisor: Mads Haahr

Dynamic game worlds are game worlds which change over time according to a given rule set. Interactive game worlds allow for users to modify the state of the game world in some way. Combining these two concepts together can result in more immersive and believable game worlds. It can also lend itself to the creation of new game play mechanics.

This project will look at how natural processes can be modelled to create believable, macro-scale dynamic game worlds with high levels of user interaction. It will discuss the different components that make a natural dynamic world and explore how they can be modelled.

A prototype combining these systems together will be designed and then implemented.

It will demonstrate how the different components of the dynamic world will work together. The prototype will be highly interactive, allowing the user to actively interact with these systems in real time and discuss the potential applications of such a system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Natural game worlds are frequent in video games. Both Unity and Unreal Engine 4 have built in tools for creating them. [4] [5] Procedural generation is also used quite reguraly to aid in the creation of large natural game worlds. This paper will discuss the creation of game worlds that are both *interactive* and *dynamic*, opposed to the largely static game worlds that the previous methods usually produce. Before discussing these terms further, they will be defined in the following section.

1.1 Dynamic, Interactive Game Worlds

An *interactive game world* is one that the player can interact with and influence. Games have varying levels of interactivity rather than being purely interactive or non-interactive. For example, many narrative focused games will limit the player's interaction with the game world in order to tell a set linear story while *god games* usually focus on player interactions with the game world or characters in that game world.

It is worth noting that while the term *dynamic* is often used when describing procedural generation, this paper is discussing continually changing dynamic elements of a game world and not dynamic generation of game worlds. A *dynamic game world*, in the context of this paper, is one which adheres to a set of rules and can change over time without necessarily relying on input from the player. This is opposed to a *non-dynamic* or *static* game world, which does not change after it is initially created.

To better understand how these terms relate to current game worlds, figure 1.1 was created to show the degree to which different games fit into these definitions.

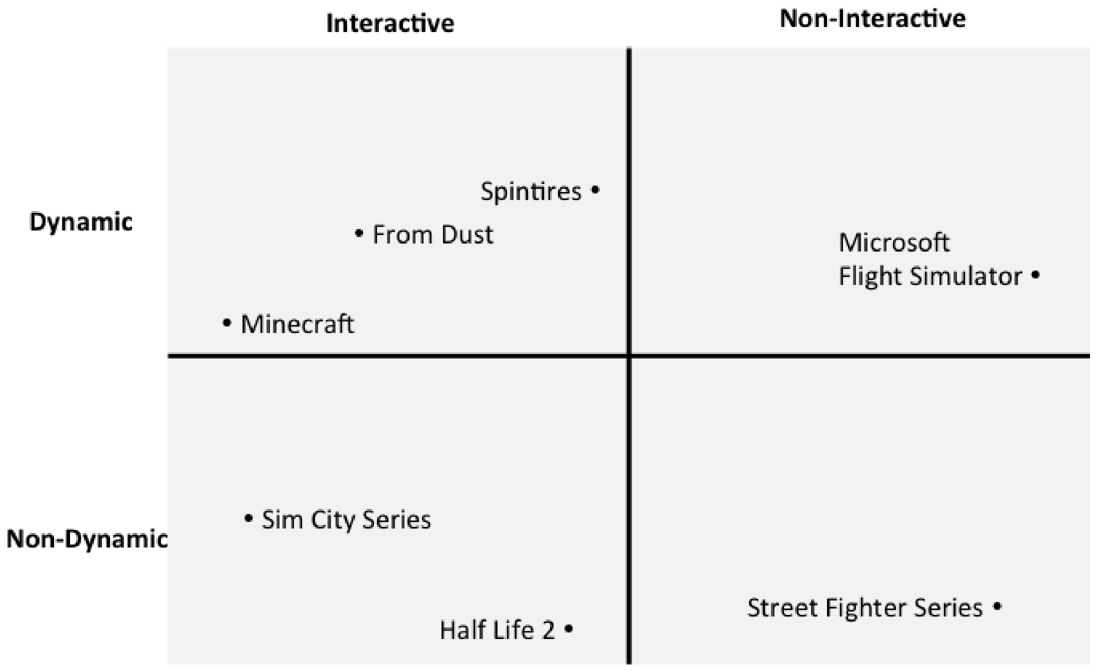

Figure 1.1: A comparison of interactivity and the dynamic nature of different games.

Game worlds which are purely static fall into the non-dynamic, non-interactive section of the chart in figure 1.1. Games such as *Call of Duty* would fall into this catagory. The terrain in the *Sim City* series could be defined as an interactive, non-dynamic system. It allows for terraforming by the player but do not have any sort of actively changing terrain. *Half-Life 2* has a non-dynamic game world, but some interactive physics objects are littered throughout it. Non-interactive, dynamic games include *Microsoft Flight Simulator* that has a dynamic weather system. This downloads real weather data and uses it in the game. Many fighting games including the *Street Fighter* series take place in non-interactive, non-dynamic game worlds, the entirety of the game is built around two character fighting in a static environment.

The interactive, dynamic game worlds section of the chart is the area this paper will be focusing on in this paper. *Minecraft* is an extremely interactive world where anything can be created and destroyed by the player but the dynamic nature of the game world is extremely limited, some blocks can fall straight down and there are some very simple fluid simulations. *Spintires*, an off-road driving simulator, has a lot of dynamic elements at play such as dynamic soil and mud deformation and dynamic water. The player can interact with these systems by driving on the terrain. *From Dust*, a god game about helping villagers traverse small islands, has a highly interactive game world, allowing players to pick up and drop soil and water. The dynamic elements of the game world are limited to the fluid simulation.

1.2 Motivation

Highly interactive and dynamic game worlds are not very common due to hardware constraints and the difficulty of scaling interactive, dynamic environments. Making these game worlds more interactive can have numerous advantages. It can help build more believable and immersive worlds for players. It can be used to create new gameplay mechanics, from the simple destructible worlds in older games such as *Artillery* or *Scorched Earth* to modern games including the god-game *From Dust* [7].

There is also the potential for interactive environment simulations to be used as tools by level designers. Having the core of the creation tools adhere a physical simulation can increase the believability and geographical coherent of the resulting game worlds.

The prevalence of multi-core CPUs and the growing trend towards GPU based physics simulations, such as Nvidia's *PhysX* [22], parallel computing is now used occasionally for more general computations in games. Previously it was rarely used in game engines outside of rendering. Leveraging this, more complicated interactions and dynamic elements can scale up more easily.

1.3 Objectives

The main goal of the project is to explore creating believable, dynamic and interactive natural worlds. There are many different processes that could make up a dynamic world simulation. This paper will focus on the following ones:

- Soil
- Water & hydrolic erosion
- Vegetation
- Wind
- Evaporation & rain

This will be achieved by completing these objectives:

- Develop models for the different processes;
- Design a system that combine these models together;
- Implement a prototype game world using this design;
- And add interactive elements to the prototype.

Other components such as wild life, snow and ice modelling were considered but were ultimately dropped due to time constraints and their contribution to the model compared to other potential processes. These are discussed more in section A.1 of the appendix.

The model must be designed in such a fashion that the algorithms driving the different processes could be easily implemented using multiple threads or using a parallel architecture.

The prototype can be used to further evaluate the project. The prototype must show how the different components can interact with each other in a meaningful way to justify simulating each of them. The prototype should demonstrate some of the uses of an interactive dynamic game world, either as a tool for level designers or a system that could be used as part of a game. While the prototype will be a proof of concept of for these models rather than a prototype focusing purely on optimising these models, it still needs to be capable of running at a reasonable speed and terrain resolution to demonstrate these models and it's potential applications.

By completing these objectives, a prototype showing the potential of such a system will be created and the complexity and scalability of such simulation can be observed.

1.4 Dissertation Outline

Chapter 2 discusses relevant research in the area of dynamic game worlds and real world research. The different potential fluid simulations, that are used extensively in this project, are also discussed. Chapter 3 discusses the chosen fluid simulation and the high level design of each of the components which make up the system. Chapter 4 discusses the implementation of each of these components and the design of the prototype. Chapter 5 evaluates the design and prototype. Capter 6 concludes the project and discusses future work.

Chapter 2

Background Research

2.1 Fluid Simulation Techniques

Fluid simulations will be used to drive some of the components of the game world. These simulations fall into one of three main categories: Eulerian fluid simulations, semi-Lagrangian, Eularian fluid simulations and Lagrangian fluid simulations.

2.1.1 Eulerian Method

Eulerian fluid simulations represent velocities and densities in discrete space as a grid of cells. The exact method used is the column model, where a two dimensional grid is used to represent columns of fluid densities. This model is often used in shallow water fluid simulations due to it's simplicity. [14] [13]

This model connects the neighbouring four cells using virtual pipes. Each cell has four out-flow pipes leading out into neighbouring cells and four more pipes leading in from each of it's neighbours as shown in figure 2.1.

Over each time step the outflow for each cell is updated. The outflow rate change is dictated by the difference in height, h between neighbouring cells, the length, l, and

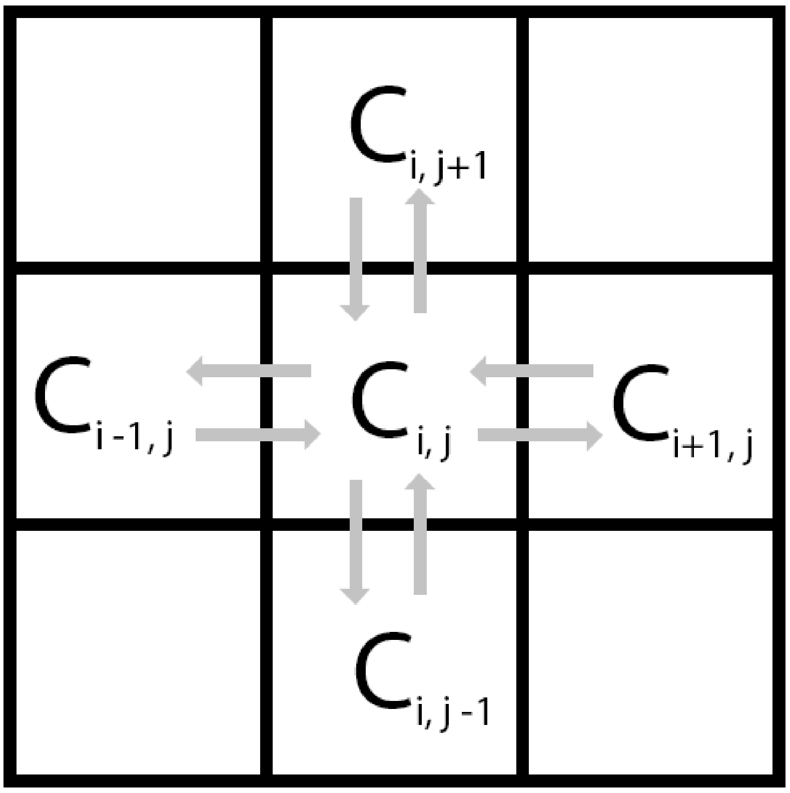

Figure 2.1: Pipe model with direct neighbours.

cross sectional area, A of the virtual pipes and gravity, g. Equation 2.1 shows the outflow calculation from cell a to b.

$$O_{a,b} = g(h_a - h_b)\frac{A}{l}\Delta t \tag{2.1}$$

This value is then added to the current outflow rate. The outflow is always kept above zero as the virtual pipes are thought to be one-directional. After the outflow rate is calculated for each direction O_L, O_R, O_T and O_B a scaling factor, K is calculated as shown in equation 2.2. [12]

$$K = \frac{hl^2}{(O_L + O_R + O_T + O_B)\Delta t} \tag{2.2}$$

If K is less than one, the outflow in each direction is scaled by K. The new heights of the cells are updated according to the outflow from the cell and the inflow from neighbouring cells. The scaling factor, K prevents the total outflow from being greater than the height of each cell so that it can never be less than zero.

This method can be implemented with $O(n^2)$ time complexity where n is the width of the two dimensional square grid. While the performance and complexity of this method is favourable, it has the disadvantage of being unstable. If the outflow rate is too high or the time step is too large, oscillations can occur within the model. Some measures need to be taken to prevent this from happening, such as limiting the maximum flow rate.

2.1.2 Semi-Lagrangian, Eulerian Method

Jos Stam proposed a stable Semi-Lagrangian, Eulerian method for fluid simulation. [17] This method also uses a grid representation but treats the movement of densities and velocities like particles to overcome the instabilities of the purely Eulerian approach.

The algorithms required for Stam's approach can be split up into three catagories: *diffusion*, *advection* and a *mass conserving* step.

For a stable model, diffusion has to be capable of working over time steps where a fluid can diffuse farther than a single cell. To achieve this, simply swapping densities between neighbours is not sufficient. Instead, a linear solver, typically using the *Gauss-Seidel method* [20], is used.

Advection is used to move densities and velocities. This step uses a *semi-Lagrangian* method as it is not merely passing densities and velocities from one cell to another. Instead, the current velocities are used to back-trace to the position where the velocity or density came from, as shown in figure 2.2. This position is not in discrete space so this method is thought of as being semi-Lagrangian. Using this position, a value is found by interpolating between the nearest cells.

The mass conserving step addresses the errors that velocity advection and diffusion adds to the simulation. Unfortunately advection of velocity is not mass conserving so

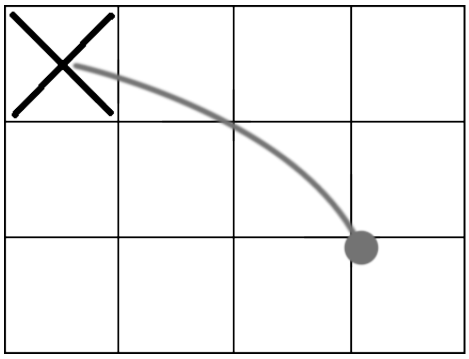

Figure 2.2: The point shown by the dot is found by tracing backwards along the velocity of the cell marked by the X by one time-step. [18]

this extra step is required. Figure 2.3 shows how to calculate this. The incompressible field is known but the gradient field needs to be calculated. This involves solving a linear system described by a Poisson equation [21] using the *Guass-Seidel method* discussed before. Once this gradient field is calculated, it is subtracted from the velocity field to find the mass conserving velocity field.

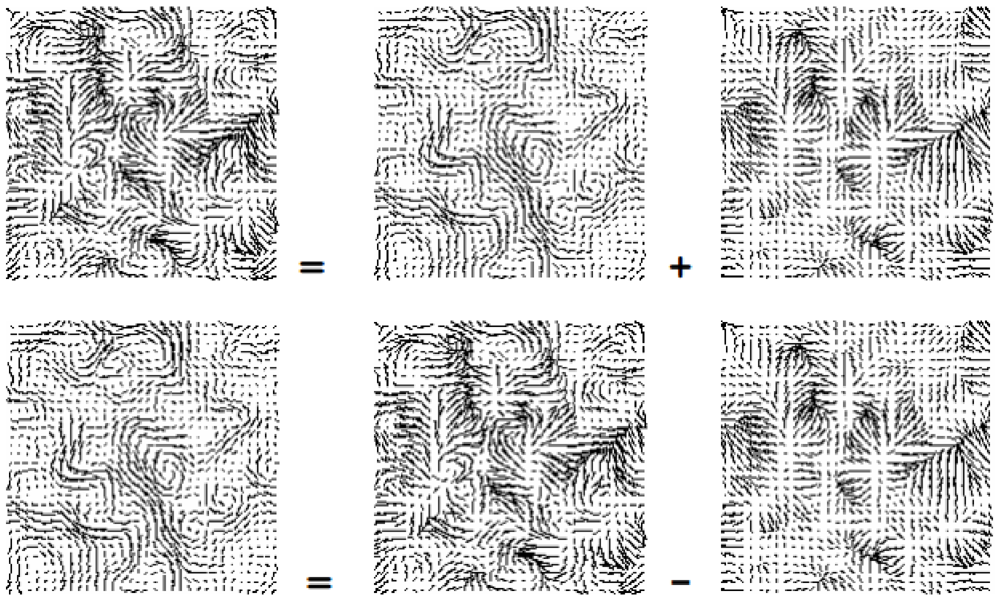

Figure 2.3: *Top:* the velocity field is equal to the sum of the incompressible field and the gradient field. *Bottom:* rearranging these to obtain the incompressible field. [18]

At the beginning of each timestep external forces are collected. The velocity step occurs first. The velocity step consists of adding new velocities and a diffusion and advection step each followed by a mass conserving step. The density step then occurs. The density step consists of adding new densities to the simulation followed by a diffusion and advection step.

This method has the advantage of being stable over large time steps. Unfortunately it has a time complexity of $O(n^2k)$, where n is the width of the two dimensional square grid and k is the complexity of the linear solver.

2.1.3 Lagrangian Method

Unlike Eulerian fluid simulations, Lagrangian fluid simulations represent bodies of water using particles. The fluid simulation is driven by interactions between these particles. These interactions describe the mass, momentum and energy conservation equations.

These models can be useful to believably simulate small volumes of water. They have the advantage of not requiring spatial connectivity information which Eulerian methods require. [15]

For our model, spatial connectivity will be important to tie the different components together. The macro-scale of the model means that small scale accuracy is not a primary concern. For use in a water simulation across a 3D terrain, a full 3D Lagrangian simulation would be required. A 2D shallow water simulation could be implemented using other methods for this purpose. It would be far less resource intensive. For these reasons, a pure Lagrangian method will not be considered for this project.

2.2 Water & Hydraulic Erosion Simulation

Two-dimensional shallow water fluid simulations can be expanded to describe water flow over a height map. The individual cells describe the height of water. In a semi-Lagrangian, Eulerian simulations, as described in section 2.1.2, gradients in the terrain and water heights can be used to add velocities to the simulation. With a pure Eulerian method, as described in section 2.1.1, terrain heights are simply added to the water heights when calculating each cell's outflow.

In the paper *Fast Hydraulic Erosion Simulation and Visualization on GPU* [12], erosion and deposition is dictated by the sediment transport capacity of the flow. This value is determined by equation 2.3 where $C_{i,j}$ is the sediment carrying capacity, $\alpha_{i,j}$ is the gradient, $\vec{v}_{i,j}$ is the velocity and K_c is the carrying capacity constant.

$$C_{i,j} = K_c . \sin(\alpha_{i,j}) . |\vec{v}_{i,j}| \tag{2.3}$$

On flat terrains the value for α can be very low. To avoid this, a user defined threshold prevents it from dropping below a certain amount.

Equation 2.4 and equation 2.5 show how deposition and erosion is calculated for each cell. h is the height of the cell, s_t is the sediment currently suspended and s_n is the new sediment value. K_s is sediment capacity constant and K_d is the deposition constant. The value for C is calculated by equation 2.3. If C is greater than s_t equation 2.4 is carried our, representing erosion. Otherwise equation 2.5 is carried out representing deposition.

$$\begin{aligned} h_{t+\Delta t} &= h_{t+\Delta t} - K_s(C - s_t) \\ s_n &= s_t + K_s(C - s_t) \end{aligned} \tag{2.4}$$

$$\begin{aligned} h_{t+\Delta t} &= h_{t+\Delta t} - K_d(s_t - C) \\ s_n &= s_t + K_d(s_t - C) \end{aligned} \tag{2.5}$$

2.3 Atmospheric Modelling

Simulating the effects of the atmosphere in games is not an area that has been greatly explored in the past. *Dwarf Fortress* is one of the few games to feature any form of weather simulation. Unfortunately it's inner workings are not well documented publicly. [1]

The *primitive equations* which drive atmospheric modelling are well documented and have been estimated using Eulerian and Semi-Langarian fluid simulations as described in section 2.1. [16]

Our atmosphere model will focus on the *evaporation*, *advection* and *precipitation* components of the *water cycle*.

Evapotranspiration is the total evaporation and plant transpiration that occurs on Earth over land, oceans and other bodies of water.

Over bodies of water, water molecules need to gain enough kinetic energy to evaporate. Factors such as wind speed, humidity and temperature will dictate the rate of evaporation. Equation 2.6 shows a method for calculating the evaporation of a body of water where g_s is the amount of water evaporated per second, Θ is the evaporation constant, A is the surface area, x_s is the maximum air humidity ratio and x is the actual humidity ratio in the air. Θ is given as $25 + 19v$ over water where v is the air speed. [2]

$$g_s = \Theta A \frac{(x_s - x)}{3600} \tag{2.6}$$

Over land, evaporation is mostly brought about by transpiration from vegetation taking water out of the soil and releasing it through their leaves. The type of vegetation will dictate the level of evaporation. Much of the information about evaporation rates for different types of forests and crops is collected through rigorous studies of specific types of vegetation under different conditions. Table 2.1 shows some of the different average evaporation rates from different types of forests.

Movement of water vapour is brought about by wind advection. This could be described

type	$\frac{mm}{hr}$
Coniferous Forest	
Scots pine	0.19
Monterey pine	0.21
Sitka Spruce	0.18
Douglas fir	0.15
Broad-Leafed Forest	
Common beech	0.45
Southern beeches	0.37

Table 2.1: Average evaporation values for canopies from different forest types. [11]

by the advection step of the fluid simulation described in section 2.1.2.

Precipitation is caused by vapour condensing back into water. Condensation generally occurs when air with a high water vapour density cools. Cooling is usually caused by rising air. [8]

2.4 Vegetation

Vegetation models are sometimes used in games, though usually in a very simple form. The game *From Dust* uses cellular automata to decide whether vegetation grows, spreads or dies. [7]

More advanced models exist outside of the game space which also use cellular automata but use it in more advanced ways. In *A Model Based on Cellular Automata for the Simulation of the Dynamics of Plant Populations*, a more realistic model is proposed. Information about plant types, resources, plant size, fruit and seed creation, size information for trunk roots and leaves is all stored. More advanced rules such as sustenance, reproduction, resource production and resource flow is built into the model. [6] With more information such as altitude and rainfall from the other components of the model, a more believable vegetation model should be possible.

There are many other ways in which vegetation could interact with the rest of the model as well. The roots of plants and trees are capable of reinforcing soil, causing steeper slopes to form. [19] They can also reduce the amount of soil lost to wind and water erosion. [10] Finally landslides and soil slip can uproot and destroy vegetation.

Chapter 3

Design

3.1 Fluid Simulation

A semi-Lagrangian, Eulerian fluid simulation can be suitable for a multi-threaded or parallel implementations but in order to maximise the resolution of the prototype a Eulerian fluid simulation will be used.

This means that the simulation will need some way of dealing with potential oscillations. The most noticeable and damaging to the believability of the model are high-frequency, low-magnitude oscillations. They cause small, visually repetitive waves to appear in the simulation.

The naive approach to prevent these oscillations occurring is to limit the maximum outflow that each cell can have. With certain threshold parameters this can eliminate these oscillations. Unfortunately, it can result in unrealistic behaviour when cells with large height differences are put side by side.

Figure 3.1 shows this scenario occurring when a large volume of water is added beside a lower volume. The volume of water on the right slowly reduces and spreads out. A visible *step* between the two volumes forms and stays in place until the water level in the two regions equalises.

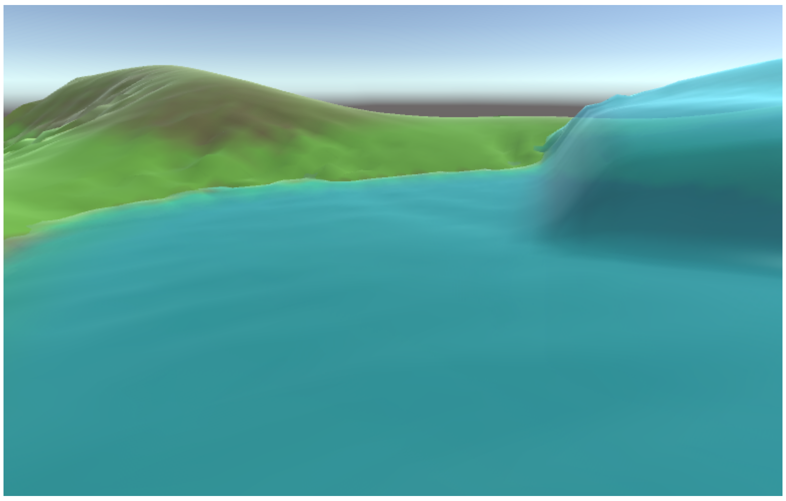

Figure 3.1: Water simulation using outflow thresholds to prevent oscillations.

To find a good solution to this problem, a simple one dimensional model was created with the aid of *Matlab*. The initial height values of this simulation are shown in figure 3.2.

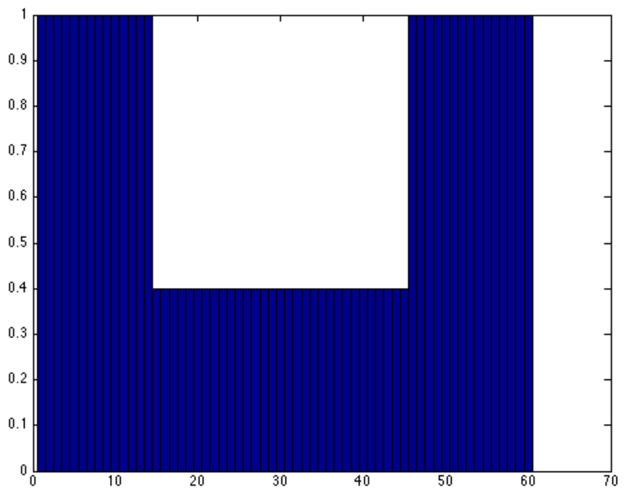

Figure 3.2: Initial simulation state.

Figure 3.3 shows the height values of the simulation using hard outflow thresholds after one second. Visible steps form throughout the simulation where there are large height differences.

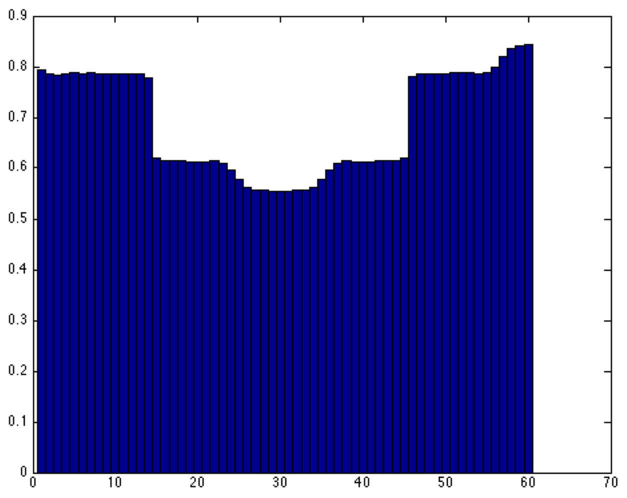

Figure 3.3: Simulation using hard thresholds.

Instead a soft outflow threshold can be created that takes into account the height differences between cells. Equation 3.1 shows how this threshold is calculated for an outflow from cell a to cell b where O_{min} is the minimum outflow threshold and s_O is the outflow scaling factor. This equation dampens low magnitude oscillations, while allowing for large volume differences to quickly and believably dissipate.

$$O_{ab} = O_{min} + s_O|h_a - h_b| \tag{3.1}$$

Figure 3.4 shows the height values after 0.3 seconds. After one second, the height values have equalised as shown in figure 3.5. Small fluctuations appear on the surface but there are no visible, repetitive, low-magnitude oscillations.

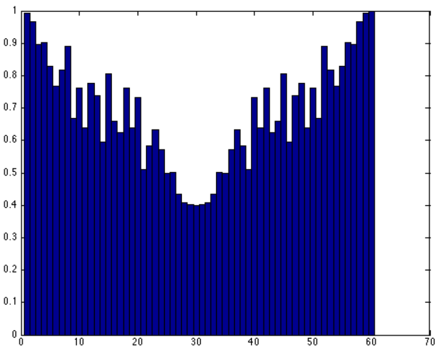

Figure 3.4: soft threshold, t = 0.3s.

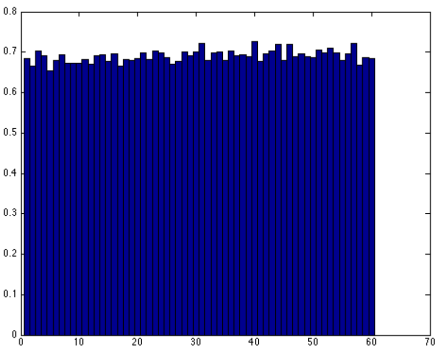

Figure 3.5: soft threshold, t = 1.0s.

3.2 Terrain Representation

3.2.1 Soil

The terrain is represented as a two dimensional matrix of soil height values. At each time step the gradient of the terrain is calculated. The gradient values are compared against a gradient threshold. If the values are higher than this threshold, some soil is transferred to the neighbouring cells. This process gives the impression of soil slip or even landslides.

3.2.2 Water

A two dimensional matrix is used to track water height values. The water simulation is driven by the fluid simulation from section 3.1. The soil height and water height matrices are added together. This results in height values for the fluid simulation that take the shape of the terrain into account.

3.3 Hydrolic Erosion

Since the fluid simulation uses a pure Eulerian implementation, information about the water velocity is not explicitly stored. To remedy this, water velocity is implicitly calculated using the height difference between neighbouring cells. Doing this produces two velocity matrices, one for horizontal velocity and another for vertical velocity. [12]

The dissolved sediment is represented by a 2D matrix. Using equation 2.3 the carrying capacity of the water is calculated and equations 2.4 and 2.5 are used in conjunction with the sediment matrix to calculate the amount of erosion or deposition if water is present.

Sediment transport is handled using a semi-Langarian method. As shown in figure 2.2, the velocity field is used to back-trace where sediment has come from. This is similar to the semi-Lagrangian step in a semi-Lagrangian, Eulerian fluid simulation. [12] For each point on the velocity matrix, this back-tracing occurs and a new array of sediment values is obtained.

Unfortunately, this method is not mass conserving and the total amount of sediment in the new array is not equal to the total amount of sediment in the old array. A linear-solver, as in the semi-Lagrangian, Eulerian fluid simulation, could be used. This would mean that the current $O(n^2)$ solution would become an $O(n^3)$ solution.

Instead a more simple mass conservation method is used. The total mass of the sediment in the matrix before sediment transport and after transport, m_1 and m_2 is calculated by summing each matrix. A scaling value k is then found where $k = \frac{m_1}{m_2}$. The values in the after-transport matrix are then scaled by k. This method is not as accurate as using a linear solver would be but for the sake of calculating the movement of sediment, it is good enough.

Finally, if there are cells which no longer contain water but have some sediment, the sediment is added back into the soil matrix.

3.4 Vegetation

The vegetation model used in the game world will not be as complicated as those discussed in the background research. The primary use of vegetation in the prototype will be demonstrating how it can interact with other game systems. This is opposed to an in-depth simulation of different types of vegetation types interacting with each other. A matrix is used to track the amount of vegetation in the model, a value between zero and one is used to signify the vegetation density of each point.

There are several driving mechanisms for the vegetation model. The first is water absorption and spreading. Water is removed from the water matrix and added to another matrix representing the absorbed or ground water within the vegetation. If there is sufficient water in a cell of the vegetation matrix, it can pass some of it on to neighbouring cells. This is similar to the method of resource transport used in in the paper *A model based on cellular automata for the simulation of the dynamics of plant populations.* [6]

The vegetation also has mechanisms for spreading, growing and dying off. If a cell has sufficient levels of water and a sufficient vegetation density, it is capable of seeding neighbouring empty cells. Some of the water is passed over to this new cell and the vegetation density is incremented by the growth rate for that time delta. Growth is dictated by a preset growth rate and requires the absorbed water in that cell to be above a certain user defined threshold. If the water falls to zero, the vegetation density starts to decrease. Another method for decreasing vegetation is through suffocation, once the water matrix goes above a certain height the vegetation dies off. This allows small amounts of water from rain to be harmless to the vegetation while allowing rivers and large other large water flows to destroy vegetation.

Interaction between the vegetation and the terrain occurs in three distinct ways that roughly model the interactions described in section2.4. The first is the prevention of soil slip. The more vegetation a cell has, the more that cell resists soil slip in the terrain simulation. This is achieved by linearly increasing the soil slip gradient threshold with the amount of vegetation present. The second is erosion prevention. This is achieved by linearly reducing the soil absorption rate of the hydraulic erosion model with the amount of vegetation present. Finally soil slip from the terrain model reduces the amount of vegetation in cells where it occurs.

3.5 Atmosphere Modelling

3.5.1 Evaporation

The amount of water vapour that has been evaporated is tracked using another matrix. Evaporation happens either from water in the vegetation model or water in terrain model.

Over bodies of water, evaporation happens in accordance to equation 2.6. The value for area is taken from the area of each cell in the model. The maximum vapour capacity of the liquid is determined by the altitude and air pressure as dictated by the wind simulation. The actual humidity is dictated by water vapour matrix. The evaporation constant is varied according to the velocity values in the wind simulation.

Given that the vegetation simulation is using a simplified model, the evaporation rate is determined by the amount of water present in the vegetation model and a minimum evaporation threshold. There are no different types of vegetation so there is no need to have a bunch of different evaporation rates. This loosely models the variation of canopy evaporation with canopy humidity. [11]

3.5.2 Wind Simulation

The wind simulation is driven by the same fluid simulation as the water simulation. Heights in the fluid simulation represent air pressure in the wind simulation. To create more believable behaviour of wind-flow around mountains and other physical barriers, a small modification is made to the fluid simulation. The virtual-pipe cross-sectional area is varied according to the gradient of the terrain. Uphill gradients result in smaller cross-sectional areas, while flats and downwards gradients result in larger cross-sectional areas.

A *prevailing wind* is added by removing air pressure from one side and adding it to the other. Velocity matrices are calculated as in section 3.3 for the sediment transport. The same method of transport and mass conservation is also used.

3.5.3 Rain Simulation

Rainfall occurs when the maximum carrying air-humidity ratio is less than the total air-vapour ratio. This value is varied by temperature and pressure. Pressure information can be taken from the wind simulation and temperature can be implicitly calculated based on the height of the terrain.

Chapter 4

Implementation

4.1 Overview

Unity was chosen as the platform to be used for creating the game world prototype. Because of it's ease of use and the wide array of built in functionality, it is a useful prototyping tool. The code driving the prototype will be written using *C#*, Unity's preferred programming language.

4.2 Dynamic World Architecture

4.2.1 Required Data Structures

As described in chapter 3, the prototype uses several two-dimensional matrices to store information about the different dynamic components. The terrain will need matrices for the soil height, the water height, the suspended sediment, the water outflow in all four directions and the velocity in the horizontal and vertical directions. The atmosphere model will need matrices for the air density or air pressure, water vapour, air outflow in all four directions and velocity in both directions. The vegetation model just needs matrices for the vegetation density and the water stored by the vegetation.

Individual cells in the outflow matrices, the sediment matrix, the water matrix from the vegetation model and the water vapour matrix are all updated according to both the current values in their matrix's previous state and other exterior values. Because there is a reliance on the previous state, two matrices are needed for each set of these parameters, one to store the old values and one to write new values to. In the prototype's *C#* implementation each set of values is represented by a *float[2][N,N]* array, where N is the width of each matrix. Two integer values are stored which point to the old matrix and one to the new matrix. There is a swap function that can be called in order to switch the two values.

Cells in the other matrices are updated in accordance to the cell's previous value and and other values outside of that matrix. Since they only rely on the value of that cell and not the entire matrix, only one matrix is needed for each of the these parameters.

In a *C++* implementation this could addressed by referencing each matrix with a pointer and having a simple macro to swap pointers.

4.2.2 Organising Data-Structures

The different components of the model, the terrain, the atmosphere and the vegetation are split up into individual classes: *DynamicTerrain*, *DynamicAtmosphere* and *DynamicVegitation*. A parent class, *DynamicWolrd*, is used to manage and combine these systems.

The naive object orientated approach to designing this system would be to put all the different data structures into their relevant classes as shown in figure 4.1.

There are many weaknesses that will result from this design. For example, the *DynamicVegetation* class has functions that need access to *waterHeight* array from the *DynamicTerrain* class. This would result in violations of the open-close rule and be a somewhat sloppy design.

Instead, all the data structures are stored in the parent class. Figure 4.2 shows how the arrays are all stored with this structure.

4.2.3 Organising Dynamic Component Logic

Child classes, shown in figure 4.2, are comprised of static functions that are used to perform the algorithms required to simulate the dynamic game world. The inputs to these functions are either references to arrays which will be modified or values which will be used to perform calculations but remain unchanged. All the functions have void return types as they are designed to modify existing data sets not compute new data sets.

DynamicTerrain

The *DynamicTerrain* class contains function which handle the soil, water and hydraulic erosion components of the game-world. Table 4.1 shows these functions, listed in the order they are called by during the terrain update step of the simulation.

The first function called is *ComputeOutflow*. It uses the water height and soil height values to calculate the outflow values in all four directions and update the current outflow values accordingly.

The *ComputeWaterVolume* function is then used to calculate the updated water height in each cell by summing the outflows and inflows together.

The *ComputeVelocities* function then uses updated water heights to implicitly calculate the velocities which the *ComputeErosionAndDeposition* and *ComputeSedimentTransport* functions can use to perform the hydraulic erosion calculations.

Finally the *ComputeSoilSlip* function is called. This uses the current soil values to calculate the new soil values. The integer values which point to the old and new buffers in the terrain parameters are swapped here so that the correct values can be read and written to by other components.

function	*inputs*
ComputeOutflow	*float*[,] outflowLeft, *float*[,] outflowRight, *float*[,] outflowTop, *float*[,] outFlowBottom, *float*[,] waterHeights, *float*[,] soilHeights, float deltaTime
ComputeWaterVolume	*float*[,] outflowLeft, *float*[,] outflowRight, *float*[,] outflowTop, *float*[,] outFlowBottom, *float*[,] waterHeights, *float* deltaTime
ComputeVelocities	*float*[,] outflowLeft, *float*[,] outflowRight, *float*[,] outflowTop, *float*[,] outFlowBottom, *float*[,] velocityX, *float*[,] velocityY
ComputeErosionAndDeposition	*float*[,] velocityX, *float*[,] velocityY, *float*[,] soilHeights, *float*[,] sedimentOld, *float*[,] sedimentNew, *float* deltaTime
ComputeSedimentTransport	*float*[,] outflowLeft, *float*[,] outflowRight, *float*[,] outflowTop, *float*[,] outFlowBottom, *float*[,] waterHeights, *float*[,] soilHeights, *float*[,] sediment
ComputeSoilSlip	*float*[,] soilHeightsOld, *float*[,] soilHeightsNew, *float*[,] soilHeights, *float* deltaTime

Table 4.1: Functions in *DynamicTerrain*

DynamicWeather

The *DynamicWeather* class contains function for simulating evaporation, wind and rain in the game world. Table 4.2 shows the different functions called during the weather update step.

The *ComputeOutflow*, *ComputeAirPressure*, *ComputeVelocities* and *ComputeVapourTransport* carry out the same parts of the fluid simulation as their counterparts do in the *DynamicTerrain* class. There are small differences in the atmosphere and water simulations so either a generalised fluid simulation class or two separate fluid simulations need to be used. Generalised solutions do not perform as well as more tailored solutions and, while generalised solutions are usually favourable, a faster tailored solu-

function	*inputs*
ComputeOutflow	$float[,]$ outflowLeft, $float[,]$ outflowRight, $float[,]$ outflowTop, $float[,]$ outFlowBottom, $float[,]$ terrainHeights, $float[,]$ airPressure, float deltaTime
ComputeAirPressure	$float[,]$ outflowLeft, $float[,]$ outflowRight, $float[,]$ outflowTop, $float[,]$ outFlowBottom, $float[,]$ airPressure, $float$ deltaTime
ComputeVelocities	$float[,]$ outflowLeft, $float[,]$ outflowRight, $float[,]$ outflowTop, $float[,]$ outFlowBottom
ComputeVapourTransport	$float[,]$ outflowLeft, $float[,]$ outflowRight, $float[,]$ outflowTop, $float[,]$ outFlowBottom, $float[,]$ waterHeights, $float[,]$ soilHeights, $float[,]$ vapour
AddPrevailingWind	$float[,] airPressure$, $Vector2$ direction, $float$ deltaTime

Table 4.2: Functions in *DynamicWeather*

tion is more appropriate here.

The *AddPrevailingWind* function adds and removes pressure from either side of the simulation in order to create a pressure difference. This gives rise to an airflow favouring one direction. The integer values that point to the old and new buffers in the weather parameters are swapped at this point.

DynamicVegetation

The *DynamicVegetation* class contains functions for simulating evaporation, wind and rain in the game world. Table 4.2 shows the different functions called during the weather update step.

The *ComputeVegetationWaterAbsorbtion* function handles taking water from the terrain simulation and absorbing into the vegetation simulation. The *ComputeGrowthAndSpread* function handles the growth and spread aspects of the vegetation simulation.

function	*inputs*
ComputeVegetationWaterAbsorption	$float[,]$ waterStored, $float[,]$ vegetationDensity, $float[,]$ waterHeights, $float$ deltaTime
ComputeGrowthAndSpread	$float[,]$ waterStored, $float[,]$ vegetationDensity, $float$ deltaTime

Table 4.3: Functions in *DynamicVegetation*

4.3 Integrating With Unity

All the systems written so far do not use any Unity specific functionality outside of some maths libraries. To complete the prototype a suitable visualisation is required, the dynamic world needs to be integrated with the visualisation and the prototype needs to handle interaction with the dynamic world.

4.3.1 Visualisation

A *SquareMesh* class was created to help visualise different components of the dynamic world. It can be attached to Unity *GameObjects* and interacted with in different ways.

On initialisation, a square grid of vertices is created and connected as a mesh. Their position along the grid is fixed but their height can be modified. A texture whose resolution matches that of the number of vertices is also created. Three different functions allow the class to be fed with different values from the DynamicWorld class.

The first function allows the square mesh to represent the solid land. Vegetation density values and soil heights are given as inputs. The heights are updated in accordance to the soil heights. The colour of the landscape varies from green to brown to represent the amount of vegetation present.

The second function allows the square mesh to represent the water. Water height and soil height values are given as inputs. The combined soil and water height is used to set the height of the vertices and the water content is used to vary the alpha values of the texture.

The final function is used for representing water vapour. Water vapour values are given as an input and the alpha values and the brightness of the texture are modified to represent water vapour in the air.

4.3.2 Connecting The Dynamic World & The Visualisation

Another class for managing the *DynamicWorld* class and integrating it with the different *SquareMesh* game objects is required. This is the *DynamicController* class and it inherits from Unity's *MonoBehaviour* class so that it can be used with Unity's Editor. From the editor the different game objects that have *SquareMesh* classes attached can be associated with it.

Unity's *Update* function is called once per frame and the *DynamicControler* class calls the *DynamicWorld*'s *Update* function at this point with a time delta equal to the time from the last frame.

4.3.3 Handling User Interaction

The *DynamicController* also monitors for user input whenever the *Update* function is called. Using Unity's built in user input handlers, the *DynamicController* can process movement inputs as well as raycast mouse clicks to positions in game world coordinates. This can be used to add or remove soil and water to exact points in the simulation or to increase and decrease the amount of vegetation.

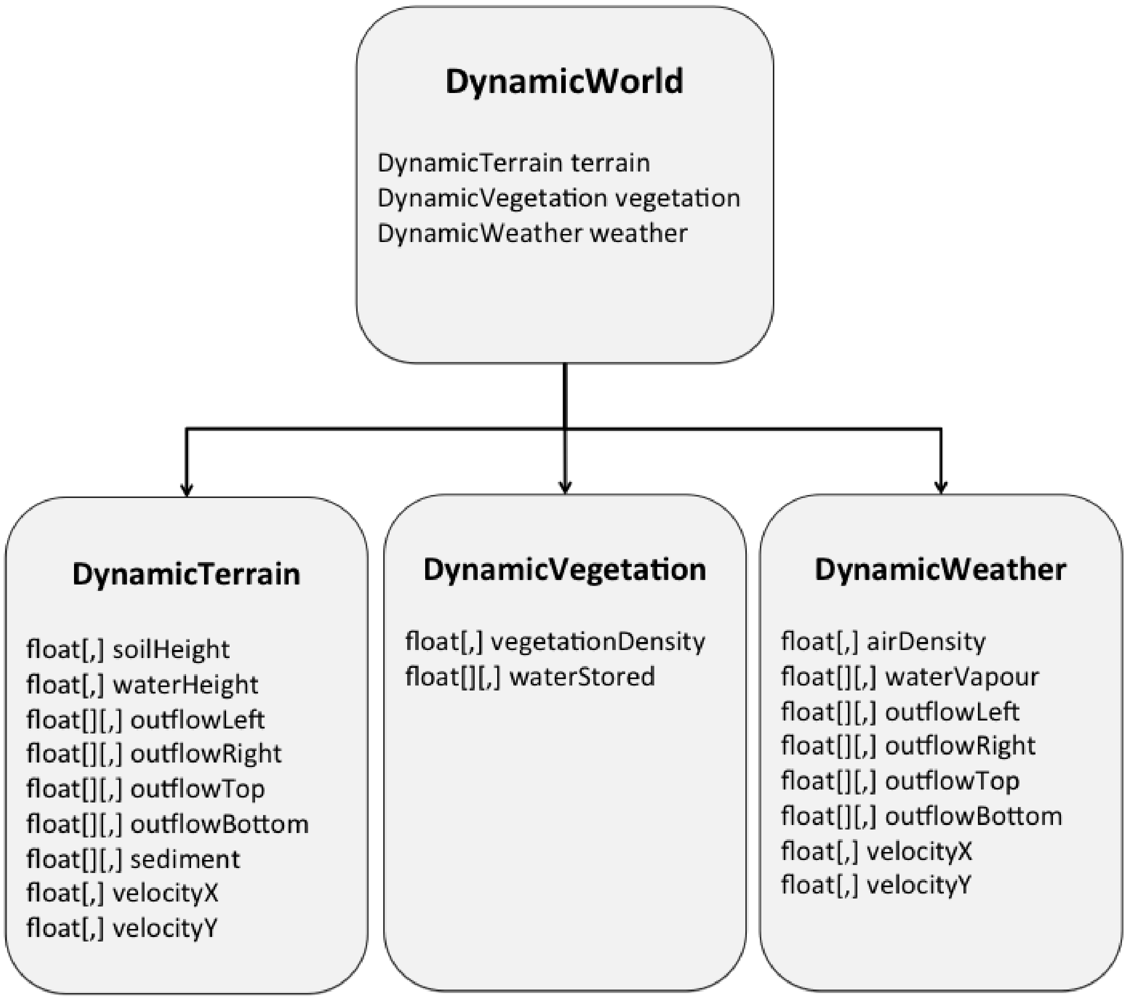

Figure 4.1: Naive object orientated approach.

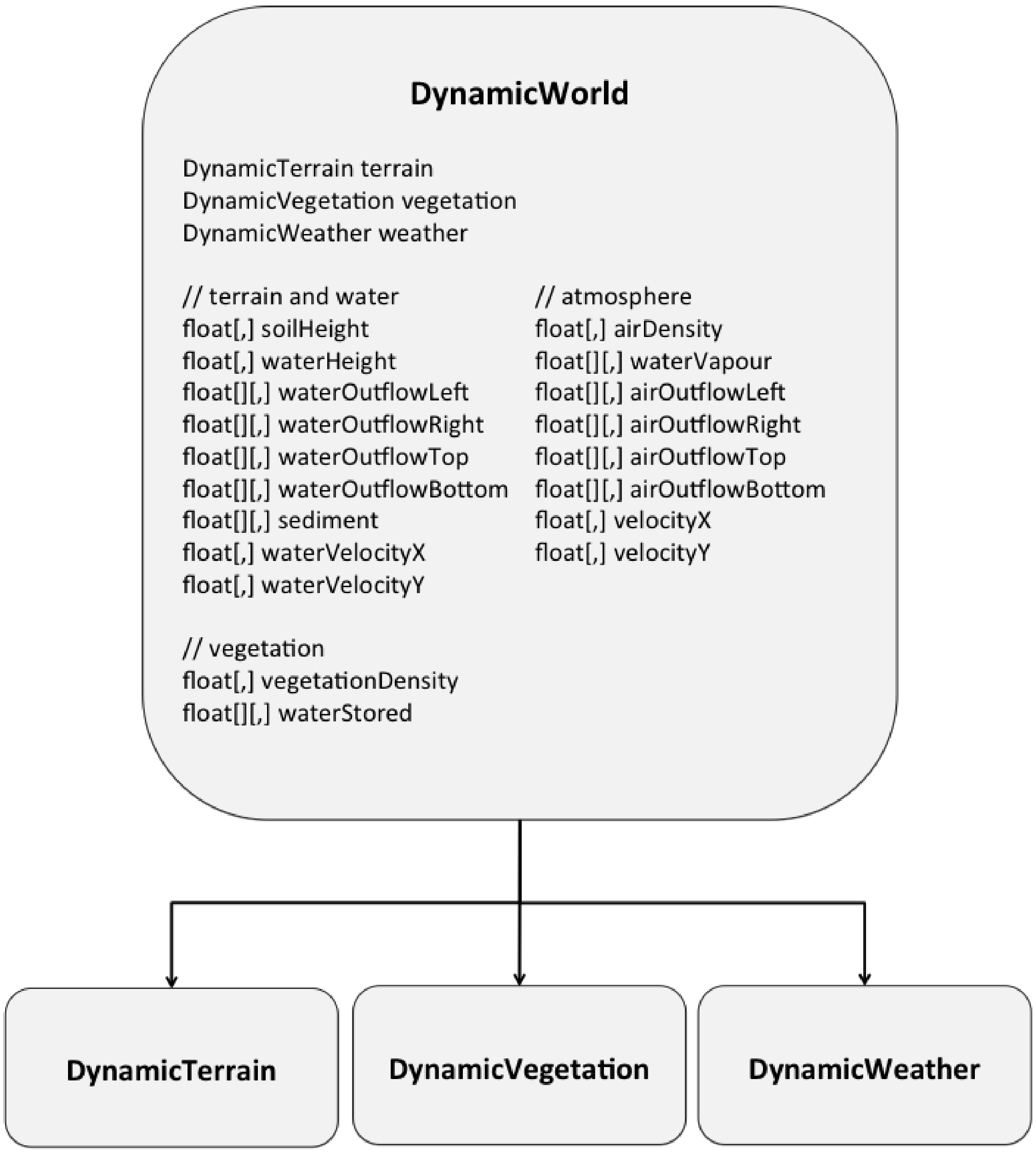

Figure 4.2: Improved hierarchy design.

Chapter 5

Evaluation

As discussed in section 1.3, the design of the systems and the prototype will be used to evaluate the project.

5.1 Suitability of Design For Parallel & Multi-Threaded Implementation

One of the design goals of our system was to create algorithms tnat would lend themselves well to parallel or multi-threaded implementations.

For a multi-threaded implementation, the matrices that make up each component of the simulation could be split into a number of smaller sub-matrices equal to the number of available threads. A final step to deal with the boundaries between sub-matrices would need to be completed. This would run in $O(n)$ time so it would not add significant overhead. All of the $O(n^2)$ operations, which are the bottleneck of our prototype, can be solved in this manner.

For a parallel implementation, all of the 2D matrices can be expressed as textures. In OpenGL the relevant algorithms are all capable of being expressed in glsl which can be executed in the fragment shader rather than the nested loops used in the CPU implementation. Compute shaders are supported in Unity which could be implemented with this method. [3]

5.2 Prototype

As discussed in section 1.3, our prototype needs to implement the soil, water, hydraulic erosion, vegetation, wind and evaporation and rain models. These models should interact with each other in believable and meaningful ways. The prototype must also demonstrate the potential for these models to be used as a development tool or game.

5.2.1 Interaction Between Different Components

The interactions between the different models is what drives the game world and as such it is important that they are believable and meaningful. This section will discus some of the different interactions that occur.

Figure 5.1 shows the formation of a river through hydraulic erosion. The top image shows the initial terrain. Water is added above the land barrier, between the existing two volumes of water. As the water begins to flow over the lowest point of the land barrier, hydraulic erosion, caused by the movement of water results soil erosion. As more water flows, the erosion increases and the river widens. Soil slip causes movement of soil towards the river which is continually eroded by it. Vegetation is killed off by the soil slip. The eroded material is deposited in the lower water volume.

Figure 5.2 shows how landslides can occur in this model. Steeper slopes can be created using vegetation that increases the soil slip gradient threshold. By removing a small amount of vegetation near the top, the soil slip gradient threshold decreases. This causes soil to move down the slope killing vegetation in the process. This has a cascading effect down the slope giving the appearance of a landslide.

Figure 5.3 shows the formation and movement of a rain cloud. Most of the evaporation takes place over land. The prevailing wind moves it over the mountains, where changes

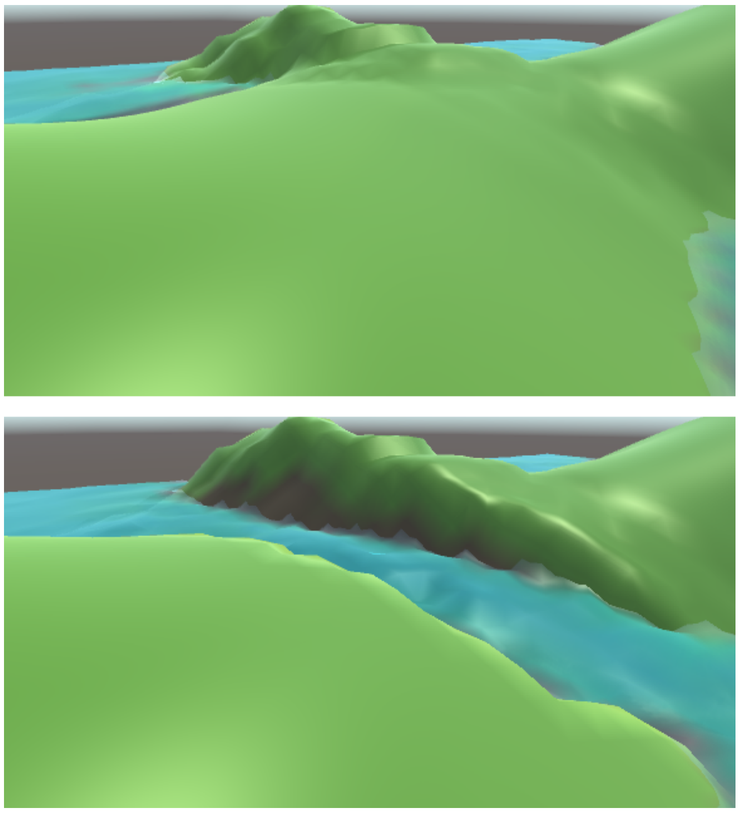

Figure 5.1: A river forming as a result of adding water above the land barrier.

in the atmosphere and elevation of the terrain model cause rain. Some of the water released can be seen in the left of the image.

Figure 5.4 shows the effect of leaving all these systems to interact with each other with no input from the user. The level of evaporation and condensation has been increased for effect. The image on the left shows the initial landscape, created using some Perlin noise. The landscape on the right shows signs of erosion, soil slip, redistributed water from the evaporation and rain and vegetation growth and death. The resulting landscape looks radically different from the initial terrain and the geographic features are more believable.

As discussed, all the different systems have their own important impact on the game world and each other. The prototype can therefore be said to fulfil the goal of simulating the different components in a believable and meaningful way.

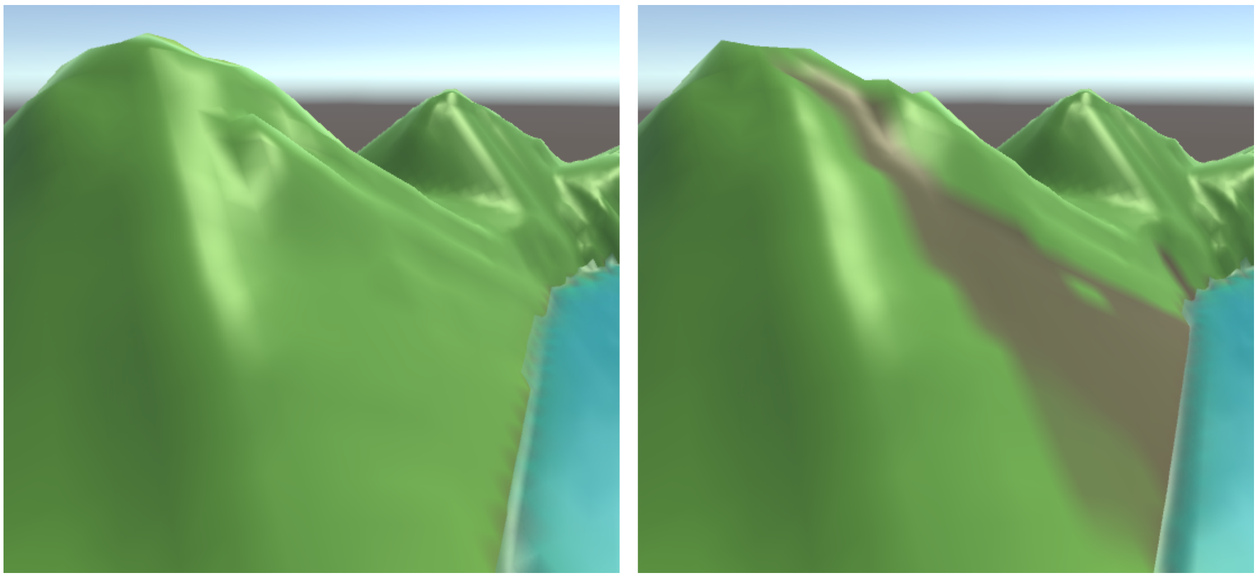

Figure 5.2: A landslide on a steep slope caused by a chain reaction of soil slip destroying vegetation.

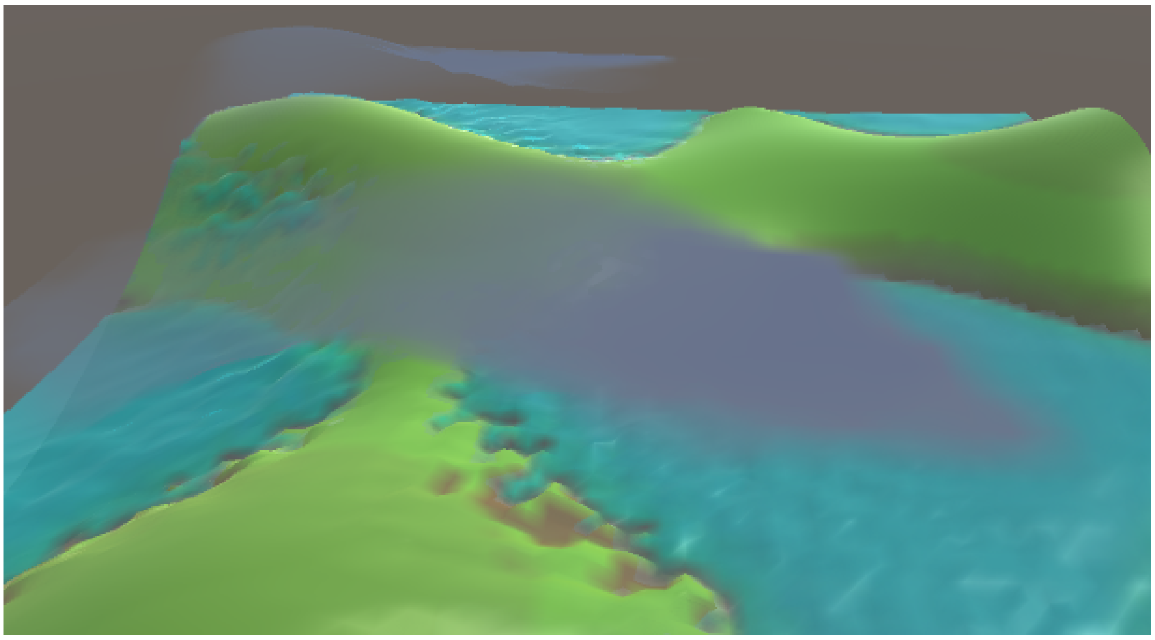

Figure 5.3: clouds moving from bodies of water up slopes where different atmosphere conditions cause them to release their stored water vapour as rain.

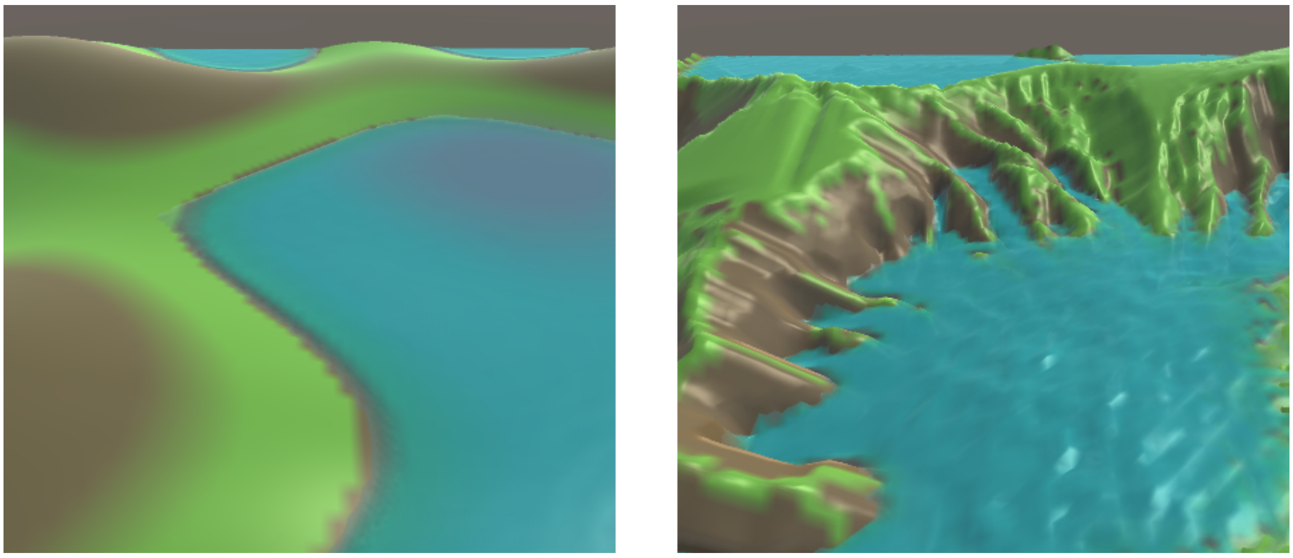

Figure 5.4: A simple Perlin noise landscape changed by only the weather simulation.

5.2.2 Potential Applications

Interactions with the game world in the prototype focus on its potential to be used as a tool for level designers.

The prototype is operated similarly to the interface of a real-time strategy game, the user can pan around the game world looking down from above and can click on the game world to interact with it. Interactions include placing and removing soil, water and vegetation. The amount of each material the user adds and the radius and size *brush* that is used to place each can be modified to allow for fine or coarse control.

A free camera mode is also available to view the terrain more closely or reposition the camera for interacting with the environment.

With these interactions, the prototype can be used as a tool to create landscapes. Since all the data comprising the world is stored in two dimensional matrices, game worlds could easily be exported for use elsewhere or loaded into our model.

The rain model could also be left to run to produce different game worlds as shown in figure 5.4.

For use in a game, the macro-scale world view and ability to manipulate the environment could lend itself well to god games, real-time strategy games or to city and town building games. The highly interactive nature and believable interactions between all the systems, discussed in section 5.2.1, could lend themselves well to an educational tool for demonstrating erosion, the effect of terrain on wind and rain and vice versa and the how vegetation interacts with the world.

5.2.3 Performance

The prototype is CPU limited. Multi-threading, while possible with the current design, was not implemented as priority was given to developing the different dynamic models. With that said the prototype is capable of running at 30 frames per second with a game world size of 110 by 110 cells on a 2012 model *MacBook Air* with a 1.8*GHz Intel i5* processor.

This frame rate is acceptable for running a tool or game. The 12,100 individual cells allowed for the different dynamic systems to be simulated at an adequate level of detail. With improvements such as multi-threading and moving away from *C#* to a more performance orientated implementation in a language such as *C++*k the performance and resolution of the system could be greatly improved.

Chapter 6

Conclusions

6.1 Contributions

The design was built using an existing fluid simulation model, soil-slip model, hydraulic erosion model and vegetation simulation model.

The soil-slip model used in other research was expanded to interact with the vegetation model, allowing for more types of behaviour including cascading landslide effects. The hydraulic erosion model was also expanded to take vegetation into account as a preventative measure of erosion.

A method for preventing oscillations from occurring in the fluid simulation was put forward that filtered out repeating small fluctuations while allowing for large movements of fluid.

An atmosphere model was proposed and implemented using the existing fluid simulation. The model used existing equations and research to simulate evapotranspiration. It used the terrain data and air pressure information to effect how wind flows. Altitude and air pressure was used to calculate the amount of condensation.

A prototype was created that combined all these models together. It was capable of believably simulating a natural game world. The prototype featured heavy user interaction, conveying the potential applications of such a simulation as a game development tool, game component or piece of educational software.

6.2 Future Work

6.2.1 Performance & Resolution

As previously discussed, the algorithms driving the prototype can be implemented with multi-threading or using a parallel implementation. This could greatly improve the resolution and performance of the system. Porting the project to *C++* and making code optimisations which are not possible in *C#* could also bring about large performance benefits.

6.2.2 Other Dynamic Components

The model contains dynamic components that are core to creating a dynamic world but many more could be added. Snow, ice, wildlife and more could be added to make a more intricate simulation. Appendix A.1 discusses some systems that fell outside of the scope of this project.

6.2.3 Different World Formats

The prototype simulates a square patch of terrain. It uses Cartesian coordinates to directly map world space back to the two dimensional matrices that hold all of the data from the different dynamic systems. An evenly spaced spherical coordinate system could be used to represent an icosphere, which is a spherical shape made up of evenly sized triangles. If these coordinates were mapped to an arbitrary array a spherical game world could be created using the same dynamic systems discussed in this paper.

By having a spherical game world, night and day could be simulated. The angle to the sun would also vary with each cell's position. The wind system could be driven by the different conditions around the game world instead of artificially creating a prevailing wind as we have to do with our square prototype.

6.2.4 Uncoupling The Dynamic Component Simulation From The Program Refresh Rate

The current implementation of the prototype directly ties the simulation of the game world to the frame rate of the program. However, this need not be the case. The last two states of the game could be stored in two buffers. These would be filled new game world states as they come out of the game world simulation. Interpolating from the oldest buffer to the newest, the game world could appear to be changing from frame to frame while being simulated at a slower refresh rate in the background.

There would be some delay between recognising a users input and it being reflected in the simulation. This could be overcome by responding visually to the users input. For example a user input to increase the height of an area could be reacted to by changing the values being interpolated to. The same values would then be added to a queue of inputs for the next simulation cycle update. The user would then still perceive their input without extra input lag and the simulation would not have to recognise it until the next update.

Changing the implementation in this way would allow for the simulation to be run on a wider array of devices with more acceptable frame rates.

Appendix A

Appendix

A.1 Other Potential Systems

A.1.1 Wildlife

Cellular automata have been used to model biological systems. The paper, *Cellular-automata-based ecological and ecohydraulics modelling* [9], explores the use of cellular automata to simulate a predator-prey model.

The vegetation model would act as a resource input to herbivore wildlife which act as the resource for the carnivore wildlife.

A.1.2 Snow & Ice

Temperature can be implicitly calculating using altitude in our simulation. When water from the atmosphere model condenses it could be modelled as snow rather than rain. The snow could be represented in a similar fashion to soil and added on top of the soil in the terrain model. Temperature variations could cause snow melt, adding water back into the terrain model. Heights of soil above a certain amount could lead to the formation of ice which could be represented by another layer.

A.2 Accompanying Documents

Attached is a disk containing the source code of the prototype and a short video of the simulation.

Bibliography

[1] Dwarf frotress wiki: Weather. http://dwarffortresswiki.org/index.php/v0.34:Weather. Accessed: 20-08-2015.

[2] Evaporation from water surfaces. http://www.engineeringtoolbox.com/evaporation-water-surface-d_690.html. Accessed: 20-08-2015.

[3] Unity documentation: Compute shaders. http://docs.unity3d.com/Manual/ComputeShaders.html. Accessed: 24-08-2015.

[4] Unity documentation: Terrain engine. http://docs.unity3d.com/Manual/script-Terrain.html. Accessed: 17-08-2015.

[5] Unreal engine 4 documentation: Landscape outdoor terrain. http://www3.geosc.psu.edu/~dmb53/DaveSTELLA/Water/global%20water/global_water.htm. Accessed: 17-08-2015.

[6] Bandini, S., and Pavesi, G. A model based on cellular automata for the simulation of the dynamics of plant populations. In *Proceedings of the International Conference on Environmental Modelling and Software Society (iEMSs)–14-17 June 2004 University of Osnabruck* (2004), Citeseer, pp. 277–282.

[7] Bel, R., and Vimont, B. Developing the interactive dynamic natural world of from dust. In *ACM SIGGRAPH 2011 Talks* (2011), ACM, p. 22.

[8] Bice, D. Modeling the global water cycle. http://www3.geosc.psu.edu/~dmb53/DaveSTELLA/Water/global%20water/global_water.htm. Accessed: 13-05-2015.

[9] Chen, Q., Ye, F., and Li, W. Cellular-automata-based ecological and ecohydraulics modelling. *Journal of Hydroinformatics 11*, 3-4 (2009), 252–265.

[10] Gao, Q., Ci, L., and Yu, M. Modeling wind and water erosion in northern china under climate and land use changes. *Journal of Soil and Water Conservation 57*, 1 (2002), 46–55.

[11] Kozlowski, T. *Additional Woody Crop Plants*. No. v. 7. Elsevier Science, 2012.

[12] Mei, X., Decaudin, P., and Hu, B. Fast hydraulic erosion simulation and visualization on gpu. In *Pacific Graphics* (2007).

[13] Müller-Fischer, M. Fast water simulation for games using height fields.

[14] O'Brien, J. F., and Hodgins, J. K. Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation* (Washington, DC, USA, 1995), CA '95, IEEE Computer Society, pp. 198–.

[15] Premžoe, S., Tasdizen, T., Bigler, J., Lefohn, A., and Whitaker, R. T. Particle-based simulation of fluids. In *Computer Graphics Forum* (2003), vol. 22, Wiley Online Library, pp. 401–410.

[16] Smolarkiewicz, P. K., and Margolin, L. G. On forward-in-time differencing for fluids: an eulerian/semi-lagrangian non-hydrostatic model for stratified flows. *Atmosphere-Ocean 35*, sup1 (1997), 127–152.

[17] Stam, J. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[18] Stam, J. Real-time fluid dynamics for games. In *Proceedings of the game developer conference* (2003), vol. 18, p. 25.

[19] Van Beek, L., Wint, J., Cammeraat, L., and Edwards, J. Observation and simulation of root reinforcement on abandoned mediterranean slopes. In *Eco- and Ground Bio-Engineering: The Use of Vegetation to Improve Slope Stability*. Springer, 2007, pp. 91–109.

[20] Weisstein, E. W. Gauss-seidel method.

[21] Weisstein, E. W. Poisson's equation.

[22] Wittenbrink, C. M., Kilgariff, E., and Prabhu, A. Fermi gf100 gpu architecture. *IEEE Micro*, 2 (2011), 50–59.