UNIVERSIDADE
LUSÓFONA

Centro Universitário de Lisboa

Escola de Comunicação, Arquitetura, Artes e Tecnologias da Informação

Mestrado em Engenharia Informática e Sistemas de Informação

Artificial Landscape Generation through Ant Colony Behaviour

Orientador:

Prof. Dr. Nuno Maria Carvalho Pereira Fernandes Fachada

Co-Orientadores:

Prof. Dr. João Pedro Leal Abalada De Matos Carvalho

Prof. Diogo Nuno Dias Mesquita Gomes de Andrade

Flávio Josefo Rodrigues Teixeira de Barros Santos | 22100771

2025

www.ulusofona.pt

Centro Universitário de Lisboa

Escola de Comunicação, Arquitetura, Artes e Tecnologias da Informação

Mestrado em Engenharia Informática e Sistemas de Informação

Artificial Landscape Generation through Ant Colony Behaviour

VERSÃO PARA DEFESA

Flávio Josefo Rodrigues Teixeira de Barros Santos | 22100771

2025

Agradecimentos

Começo por expressar os meus agradecimentos à Universidade Lusófona por me ter proporcionado os recursos com os quais fui capaz de evoluir academicamente. Destaco igualmente todos os professores, tanto da Licenciatura como do Mestrado, com quem tive a oportunidade de trocar impressões e cujas interações foram sempre agradáveis e enriquecedoras.

Um agradecimento aos meus orientadores João Carvalho e Diogo de Andrade por todo o apoio dado ao decorrer deste projeto, com um especial agradecimento ao professor Nuno Fachada pela sua constante disponibilidade, paciência e por me ter aberto horizontes, tanto no que toca a este trabalho, como ao conhecimento transmitido durante a Licenciatura.

Agradeço ainda aos meus amigos Leandro Brás e José Rodrigues, com os quais tenho sempre o prazer de conviver e me proporcionam confiança, troca de ideias e diversão, experiências que dificilmente iria encontrar de outra forma.

Agradeço também à minha família, em particular ao meu Pai, Mãe e Irmã, que todos os dias me inspiram e incentivam a continuar com o meu percurso de vida. Um agradecimento final ao meu gato Kit, que, embora nunca possa ler estas palavras, merece o meu reconhecimento por ter a capacidade de tornar os momentos difíceis em mais leves e os bons em ainda melhores.

Resumo

Esta dissertação apresenta um algoritmo inovador de Geração Procedimental de Conteúdos baseado no comportamento de formigas, permitindo a criação de novas paisagens virtuais, bem como modificações realistas de paisagens existentes. Para tal, adapta as capacidades de determinação de percursos das formigas com base em feromonas e o seu comportamento de recolha de alimentos, em combinação com perceção de altitude. As formigas navegam no seu ambiente identificando pontos de interesse, tais como fontes de alimento e a sua colónia, deixando rastos de feromonas à medida que se deslocam. Estes rastos servem como sinais que atraem outras formigas, orientando os seus caminhos e permitindo uma interação colectiva. Neste processo, podem influenciar e modificar o terreno, criando caminhos ou padrões que refletem a sua atividade. O projecto foi desenvolvido com recurso ao motor de jogo Unity, tirando partido das suas funcionalidades relacionadas com terrenos. O sistema concebido suporta a criação e armazenamento de variáveis parametrizáveis que permitem a personalização do comportamento das formigas, a inicialização do terreno e algumas outras configurações relacionadas com a visualização. Um conjunto limitado de resultados demonstra a capacidade do algoritmo para se adaptar e transformar vários tipos de paisagens. De um modo geral, este projeto pretende servir de base a uma convergência de ideias ainda não explorada: unificar os conceitos de simulação de animais e criação de paisagens virtuais interessantes e dinâmicas.

Palavras-chave: Geração Procedimental de Conteúdos, simulação de formigas, geração de terreno, paisagem virtual, desenvolvimento de videojogos

Abstract

This dissertation presents an innovative Procedural Content Generation algorithm that creates new virtual landscapes, inspired by ant behaviour, as well as realistic modifications to existing landscapes. It does so by adapting ants' pheromone-based pathfinding capabilities and food collection behaviour combined with height perception. Ants navigate their environment by identifying points of interest, such as food sources and their colony, leaving pheromone trails as they move. These trails serve as signals that attract other ants, guiding their paths and enabling collective interaction. In this process, they can influence and modify the terrain, creating paths or patterns reflective of their activity. The project was built with the Unity game engine, taking advantage of its terrain generating and visualization tools. The system developed supports the creation and storage of parameterisable variables that allow the customization of the ants' behaviours, the inception of the terrain and some other display related settings. A select set of curated results demonstrates the algorithm's capacity to adapt to and transform various types of landscapes. Overall, this project's intent is to serve as the basis of a yet unexplored convergence of ideas: to unify the concepts of animal simulations and creation of interesting and dynamic virtual landscapes.

Keywords: procedural content generation, ant simulation, terrain generation, virtual landscape, video game development

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Procedural Content Generation (PCG) is an algorithm-assisted method of creating virtual content [33]. In real-time applications, such as video games, it can be used as a tool to generate anything from landscapes or characters to levels or quests. Given the current complexity of asset creation for video games, PCG can not only shorten development cycles, but also help reduce a project's monetary cost [41]. PCG can additionally be employed as a way to dynamically generate content as a player interacts with a game, increasing both engagement and replay value [31].

However, PCG is not a novel concept, with its origins dating back to the early days of video game development. Video games such as *Beneath Apple Manor* [39] from 1978, *Rogue* [35] from 1980, and *Elite*, released in 1984, used PCG as a way to address memory limitations. In recent times, video games such as *No Man's Sky* [15] market themselves with PCG techniques at the forefront, enabling the generation of the majority of scene elements through runtime algorithms (see Figure 1.1).

Many PCG algorithms have been developed to generate a wide variety of content, including textures, models, or maps [18, 2, 41]. Among these, landscape and terrain generation algorithms stand out, as they play a pivotal role in the creation of vast, immersive virtual worlds. Erosion simulation, fractal noise methods, and voronoi diagrams are some of the better known and widely used techniques when it comes to automatic landscape creation [27]. Nonetheless, search-based algorithms, also known as metaheuristic algorithms, have seen a growing interest in procedural content generation [34], offering similar results through a generate-and-test approach.

Metaheuristic algorithms are problem-independent optimisation techniques designed to address complex challenges not easily solvable through traditional heuristics or mathematical programming methods [1]. Usually inspired by natural processes such as animal behaviour or genetic evolution, these methods rely on iterative trial and error approaches, allowing for the exploration of vast solution spaces effectively. In problems where exact solutions are computationally impossible, they can often find near-optimal outcomes [6].

This dissertation builds upon the work employed by de Andrade et al. [5], where a metaheuristic algorithm known as Particle Swarm Optimisation (PSO) [22] was used to generate aesthetically pleasing 3D animations by running the algorithm over artificial landscapes. More importantly for this thesis,

Figure 1.1: A screenshot of a planet's landscape in *No Man's Sky*. Every element within the frame is a product of PCG algorithms. Source: Hello Games [15].

their work also explored the application of PSO particles on Perlin-generated terrains [17, 24], revealing only the portions of the terrain visited by the particles. While the particles themselves did not directly generate the terrain, their influence over it led to the display of dynamic and visually appealing results. Building on this foundation, this dissertation aims to advance the concept by investigating the application of an alternative swarm algorithm for procedurally generating natural and realistic landscapes, further enhancing the relationship between swarm behaviours and terrain formation.

Given ants' inherent connection to terrain-based activities, the Ant Colony Optimisation (ACO) algorithm [7] presents a compelling foundation for PCG methods. By leveraging ant-like agents capable of navigating and modifying landscapes through pheromone deposition and detection, ACO offers a natural mechanism for influencing terrain structures. These agents, traditionally reliant on graph-based movement, can act as drivers for dynamic terrain modification, generating interesting patterns along their paths. Consequently, this thesis raises the following research question: "Can ACO serve as a viable inspiration in the context of terrain PCG algorithms?"

Ultimately, however, traditional implementations of ACO are quite limited in terms of ants' ability to fully explore and interact with a three-dimensional environment. Conventional ACO approaches lack height perception, a critical capability for navigating and modifying terrains where elevation is a significant factor. Terrain modifications, such as creating realistic slopes or valleys, require an algorithm capable of accounting for height variations to maintain coherence and plausibility. These limitations suggest the need for an original swarm-based approach tailored specifically to terrain navigation and modification challenges: the *Landscaper Ants* algorithm. Thus, this thesis explores another critical research question: "How can ant-inspired behaviours be adapted to enhance the coherence and realism of procedurally generated terrain features?"

This document is structured as follows. Chapter 2 provides an in-depth review of the state of the art regarding different virtual landscape generation techniques. Chapter 3 details the methodology employed

in this project, including a breakdown of both the ACO and Landscaper Ants algorithms, alongside an overview of the software architecture and parameterisation used in the conducted experiments. Chapter 4 presents the key results, showcasing a series of experiments and emphasizing the main differences in input configurations. Chapter 5 offers a comprehensive discussion of the obtained results, addressing insights from both the perspectives of virtual landscape generation and ant simulation. It also explores the current limitations of the system and proposes potential improvements. Finally, Chapter 6 concludes the dissertation by highlighting the achievements and offering suggestions for future research.

Chapter 2

Background

Digital landscapes are a core component of many video games, as they serve as the primary spaces where players engage with missions, quests, and the overarching game experience. However, given that some video games' maps can occupy hundreds of squared kilometres, manually creating terrains can be a time-consuming task [30, 21]. The exploration of algorithmic processes to accelerate digital terrain creation has been steadily advancing, with PCG emerging as a central approach in this field. PCG is an algorithmic approach for creating environments, textures, and phenomena in digital media. It allows, for example, the creation of cohesive terrain, adhering to specific game design choices through the use of parameterizable variables. This allows for both artists and game designers to worry less about the tools at hand and instead focus on the design process of map creation.

Over the years, numerous terrain generation techniques have been developed and refined, each with its own strengths and weaknesses [27, 10]. Among these, Perlin Noise, Erosion Simulation, Midpoint Displacement, and Voronoi Tesselation stand out as key methods in PCG for terrain generation.

Perlin Noise [28] is a gradient noise function commonly used to create smooth and continuous variations in visual data. It ensures coherence across spatial dimensions, such as 2D or 3D landscapes, and maintains consistency over time for applications like animations or evolving patterns. Its application in terrain generation allows for the creation of natural-looking environments with seamless transitions, making it ideal for simulating features such as hills, valleys, and other gradual changes in elevation, as illustrated in Figure 2.1. However, despite this and its computational efficiency, Perlin Noise can produce repetitive artifacts at large scales and is not easily adaptable to some video game terrain scenarios, as discussed by Gürler & Onbaşioğlu [17]. In particular, these authors explored methods for applying Perlin Noise to terrains constructed with hexagonal tiles, addressing challenges related to seamless transitions and reducing visible repetition.

Erosion Simulation methods add realism to procedurally generated terrains by mimicking the forces that shape real-world environments. Techniques such as hydraulic erosion (caused by water flow) and thermal erosion (caused by temperature variations and gravitational forces) are akin to Cellular Automata (CA) solutions, as they evolve grid-based systems over discrete time steps according to simple, local rules [32]. Hydraulic erosion, as illustrated in Figure 2.2 simulates water flow across a heightmap,

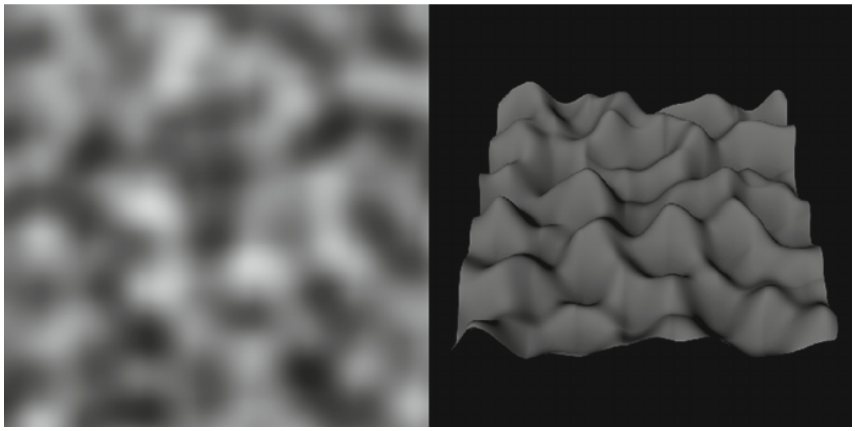


Figure 2.1: An example of a Perlin noise greyscale texture (left) and its resulting terrain (right). Source: Eck & Lamers [9].

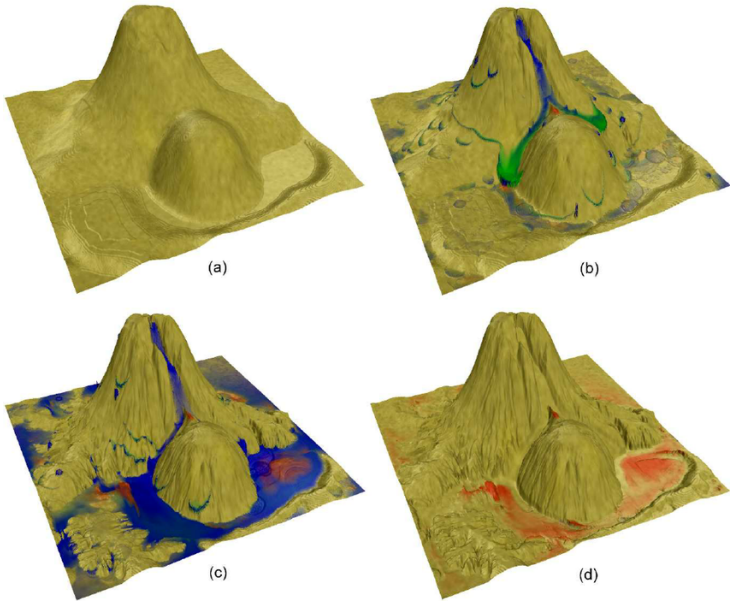


Figure 2.2: An erosion process that combines rainfall and a river source. Source: Mei et al. [25].

where water particles erode material from higher elevations and deposit sediment in lower areas, creating features like rivers and valleys. Thermal erosion, on the other hand, models the crumbling of steep slopes due to temperature-induced stress, smoothing out sharp features over time. Some works such as those from Roudier et al. [29] and Beneš & Arriaga [4] showcase efficient algorithms that use erosion simulations to create both general landforms or specific terrain formations such as table mountains. Other authors demonstrate approaches not based on natural processes, rather focusing on the use of automatons (CA-based rulesets) which can produce believable and aesthetically pleasing 3D landscapes. For example, Mikedakis [26] showcases the use of CA in the creation of three-dimensional polygon meshes through voxel-based spaces. Ziegler & Mammen [42] focus instead on generating heightmaps through CA-based rulesets, smoothing out the results with thermal erosion. Fachada et al. [11] also explore the creation of heightmaps, by iteratively adding grid states' values after applying specific transition rules and normalizing the final values, thus creating aesthetically interesting landscapes.

Midpoint Displacement algorithms [3] represent fractal-based approaches often employed in real-time terrain generation [40]. They consist in recursively subdividing a terrain space into smaller segments, displacing their midpoints by random values, effectively smoothing out the result. In the creation of two-dimensional heightmaps (which are interpreted as three-dimensional terrains), the Diamond-Square algorithm [19, 14, 23] is the *de facto* implementation of this technique. This algorithm starts by setting the values of all cells on a square 2D matrix to 0, with the exception of its four corners, which are set to random values within a chosen range. Afterwards, the following steps are performed:

1. *Diamond Step*: After finding the midpoint of the four corners (i.e., the most central cell in the matrix), set its value to be the average of the corners, plus some random value.

2. *Square Step*: After finding the midpoint of any two corners (i.e., the central cell of any previous 'diamond'), set its value to be the average of those same corners, plus some random value.

These steps are repeated recursively until the resolution limit of the matrix is reached, while also decreasing the range of the added random values. The result is a heightmap with varied elevations that can be used to render hills, mountains, and valleys. Figure 2.3 showcases the previously described process.

Voronoi diagrams are a mathematical structure that divides a plane into regions based on proximity to a set of seed points, with each region containing all the points closer to its seed than to any other. In terrain generation, Voronoi diagrams are used to create natural-looking features such as mountains, valleys, or biome distributions by treating the regions as different terrain features. By assigning varied elevations or terrain types to the seed points and interpolating these values across the regions, Voronoi-based methods produce terrain that mimics the irregular, clustered appearance of natural landscapes. Additionally, as exemplified by Olsen [27], and displayed in Figure 2.4, Voronoi diagrams can be enhanced with techniques like perturbation or noise to introduce further irregularity, adding realism to the generated terrains.

In the last few years, other methods of terrain synthesis have also been explored. For instance, Diffusion Limited Aggregation (DLA) [8], a type of fractal aggregation algorithm, has seen some use in generating mountainous landscapes when combined with image processing, as its creation derives directly from the evolution of natural structures [38].

Search-Based Procedural Content Generation (SBPCG), a subdivision of PCG methods, distinguishes itself through its extensive reliance on search algorithms [34]. These include evolutionary and metaheuristic approaches, which are designed to address complex optimisation challenges through iterative processes. For example, algorithms such as Bee Colony Optimisation (BCO) [20], Ant Colony Optimisation (ACO) [7], and Particle Swarm Optimisation (PSO) [22] offer unique approaches in dealing with optimisation problems by drawing inspiration from natural behaviours observed in bees, ants, and flocks of birds, respectively [16, 37]. These bio-inspired heuristic algorithms have seen some use in 'landscape generation', as showcased in preliminary research by de Andrade et al. [5], where PSO-based agents can navigate and display visited portions of Perlin-generated terrains, effectively transforming the original landscape (see Figure 2.5). As another example, Fernandes [12] created a method named *pherographia* based on the

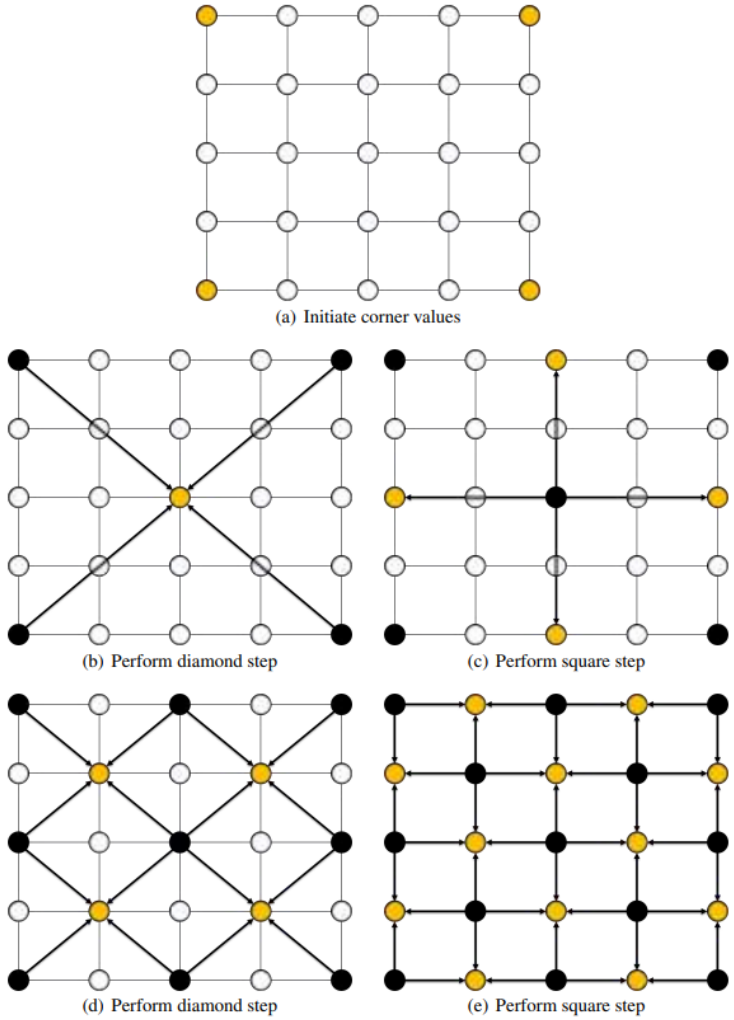


Figure 2.3: Procedure for the Diamond-Square algorithm in 5 steps. Source: Yannakakis & Togelius [40].

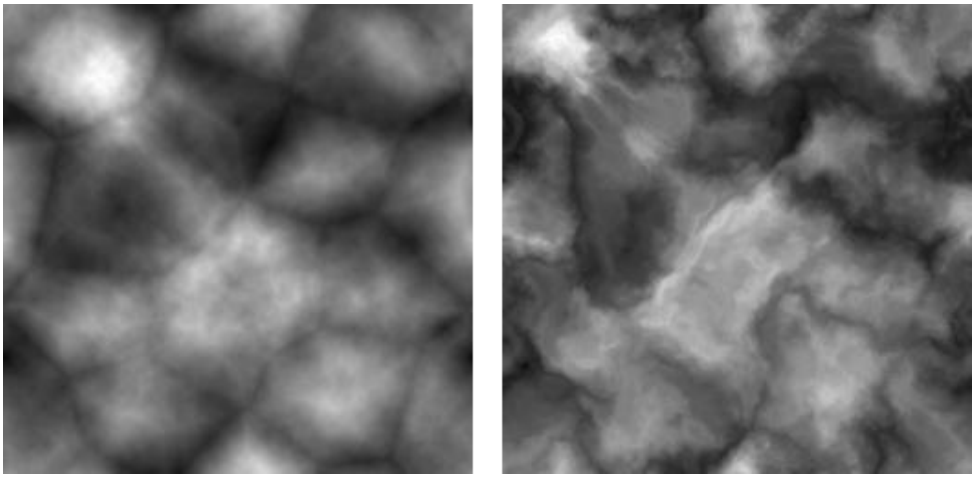


Figure 2.4: A heightmap created by combining the Diamond-Square method with a section of a Voronoi diagram (left), alongside its perturbed version (right). Source: Olsen [27].

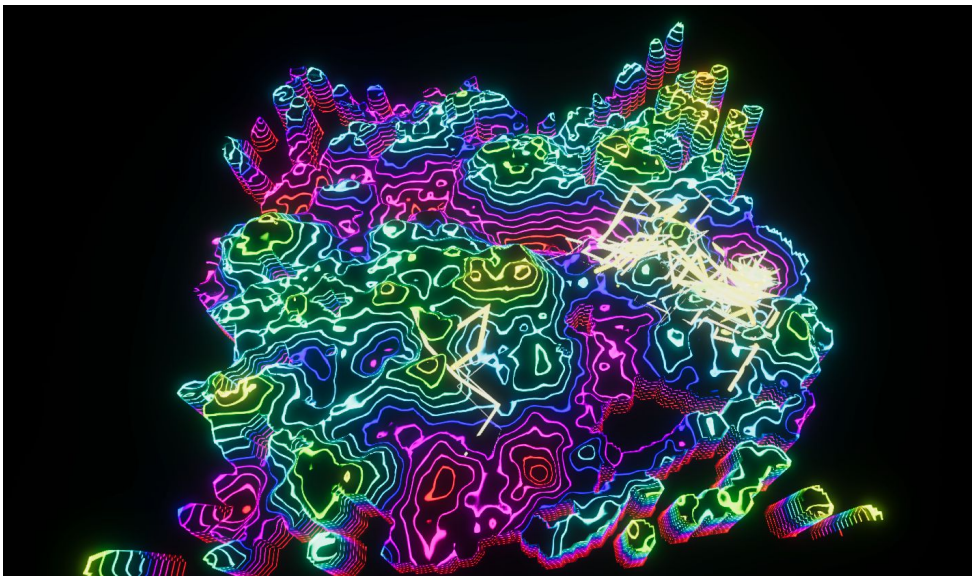


Figure 2.5: Artistic landscape created by applying PSO on a Perlin-generated terrain. Source: Andrade et al. [5].

ACO algorithm, which can create camera obscura-like drawings of supplied images. Fernandes et al. [13] later expanded the previous work, adding in the element of color (appropriately naming the method *color pherographia*). While these examples focus on algorithmic art, their principles align closely with terrain generation, where 2D matrices, or heightmaps, store values that are rendered into visual landscapes.

The methods proposed in the next chapter integrate PCG and biologically-inspired metaheuristic algorithms to create or enhance terrain. Drawing from de Andrade et al. [5], this work is inspired by the way agents interact with and influence virtual landscapes, as well as techniques such as ACO, which model natural behaviours like pheromone tracking using mathematical abstractions. By adapting these concepts, the proposed approach aims to create interesting terrains, adding detail and natural complexity.

Chapter 3

Methodology

This chapter outlines the methodology employed throughout the research. It begins by introducing the primary algorithm, offering a comprehensive explanation of its parameters and core execution methods. The next section briefly explores an ACO-based approach that, while yielding interesting results, was only utilized during the early stages of the project. Finally, the chapter concludes with an overview of the project's implementation in the Unity game engine, detailing the software architecture and introducing the concept of experiments.

3.1 The Landscaper Ants Algorithm

The primary algorithm developed in this project, referred to as *Landscaper Ants*, introduces a technique for landscape PCG where terrains are generated or modified as a result of surface exploration by agents exhibiting ant-like behaviours. These agents, also referred to as *ants*, are randomly distributed across the landscape and tasked with locating *food* sources to deposit back at their *colony*. By assessing pheromone concentrations and slope gradients, they effectively navigate the terrain, leaving behind tracks that emerge from their movement patterns. This behavior is further enhanced through a comprehensive set of configurable parameters provided by the algorithm, allowing users to customise essential structures, control ants' pathfinding decisions, and adjust the evaporation and diffusion rates of deposited pheromones. Algorithm 1 outlines the algorithm's framework, with the following subsections offering a detailed examination of its core execution methods and parameter interactions.

Algorithm 1 LandscaperAnts

```
1: InitStrucures()
2: for every step until maxSteps do
3:     UpdateAnts()
4:     UpdatePheromones()
5: UpdateTerrain()
```

Parameter	Description
`useSeed`	Boolean. If true, the random number generator uses a user-defined seed; if false, a seed is automatically selected by the system.
`rndSeed`	Integer. Defines the seed value for the random number generator.

Table 3.1: Parameters used in the setup of the random number generator.

Parameter	Description
`baseDim`	Integer. The `x` and `y` dimension values of the height and pheromone matrices.
`flatTerrain`	Boolean. If true, the terrain starts flat; if false, it begins with a Perlin Noise-based heightmap.
`foodAmount`	Integer. The amount of `Food` to generate.
`maxFoodBites`	Integer. The maximum number of interactions ants can have with any `Food`.

Table 3.2: Parameters used for the creation of terrain-based elements.

3.1.1 Terrain and Ants Initialization

The first stage of the *Landscaper Ants* algorithm is responsible for setting up all necessary structures, some of which rely on random number generation. Table 3.1 presents the parameters that allow users to control the pseudo-randomness involved in initializing these structures, as well as in later functionalities such as ants' movement patterns, ensuring the reproducibility of results. The `rndSeed` parameter defines the initial seed for the random number generator, while the `useSeed` boolean gives users the option to either use that seed or let the system generate one automatically.

After setting up the randomization parameters, the algorithm proceeds to initialize the terrain's core structures. This process begins with the creation of the terrain's height and pheromone concentration 2D matrices according to user-defined parameters, outlined in Table 3.2. These include `baseDim`, which determines the dimensions of both matrices, and `flatTerrain`, which specifies whether the surface's initial height data is flat or Perlin noise-based. The height matrix acts as both the ant's virtual environment and the terrain's heightmap, ensuring that any changes made by ants are directly reflected in the landscape visualization. Meanwhile, the pheromone matrix enables ants to deposit and assess pheromone levels throughout the terrain.

Ants can also interact with points of interest scattered throughout their digital environment, specifically food sources. These are simple constructs, defined by a location and a maximum number of interactions, and they are randomly placed within the terrain's boundaries. As detailed in Table 3.2, users can configure the number of food sources and the maximum interactions per source using the `foodAmount` and `maxFoodBites` parameters, respectively.

The presence of ants in the terrain is represented as a two-dimensional vector, similar to their shared colony, with their quantity determined by the `nAnts` parameter. Since manually positioning thousands of ants would be impractical, the algorithm allows users to choose between two initial placement options: distributing them randomly across the landscape or positioning them at their colony, controlled by the

Parameter	Description
`nAnts`	Integer. The amount of ants to be created.
`individualStart`	Boolean. If true, ants' starting positions are random; if false, they all start at the colony.

Table 3.3: Parameters responsible for the creation of ants.

`individualStart` parameter. Table 3.3 provides an overview of these ant-related parameters.

3.1.2 Ant Path Selection

Ants' pathmaking abilities serve as the primary mechanism for terrain modification. These methods establish a reciprocal relationship between the ants and their environment, enabling the ants to influence and adapt to the landscape dynamically. By making use of several *navigation cues*, ants shape the terrain as they navigate, creating pathways, altering elevations, and ultimately contributing to the emergence of complex, structured landscapes. This interplay ensures that the terrain evolves organically in response to the ants' collective behavior.

Algorithm 2 provides a pseudo-code overview of the method governing the ants' movements. The process begins by determining whether the ant array needs to be shuffled (line 1). If shuffling is required, the `Shuffle()` function (line 2) rearranges the order of ants in the array, helping to prevent a small subset of ants from dominating others' path formation.

Algorithm 2 UpdateAnts

```
1:  if shuffleAnts then
2:      ants.Shuffle()
3:  for every ant do
4:      currentCell ← ant.CurrentCell
5:      nextCell ← none
6:      neighbours ← GetMooreNeighbours(currentCell, antsInPlace)
7:      if ant.HasFood() then
8:          if ant.IsAtColony() then
9:              ant.Food ← none
10:             nextCell ← currentCell
11:         else
12:             nextCell ← GetNextPoint(neighbours, currentCell, ant.ColonyCell)
13:             pheromoneValue ← grid.Pheromones[currentCell] +
                    ant.DropPheromone(pheromoneDeposit)
14:             grid.Pheromones[currentCell] ← Clamp(pheromoneValue, 0, 1)
15:     else
16:         if FoundFood(neighbours, out food) and food.HasBitesLeft() then
17:             food.TakeABite()
18:             ant.Food ← food
19:             nextCell ← currentCell
20:         else
21:             next ← GetNextPoint(neighbours, currentCell)
22:     Dig(currentCell, nextCell, antIndex)
23:     ant.CurrentCell ← nextCell
```

The function then enters a loop where every ant is iterated over (lines 3-23). However, before any

path selection, 3 local variables are created. The first two variables, `currentCell` and `nextCell`, are two-dimensional vectors that represent an ant's current and next positions in the available space. The third variable, `neighbours`, is an array of two-dimensional vectors representing the cells surrounding an ant's current position, populated by a function that calculates the Moore neighborhood of the cell. The `antsInPlace` variable, passed as an argument, determines whether the central cell is included, allowing ants to consider their current position when selecting the next cell.

The next steps (lines 7-21) are a small collection of different nested `if-else` instructions that allow ants to select their next cell. The present conditions are:

1. The ant is carrying food and has reached its colony.
2. The ant is carrying food but has not yet reached its colony.
3. The ant is not carrying food but has found some and this food can still be interacted with.
4. The ant is not carrying food, hasn't found any, or, if it has, it can no longer be interacted with.

In conditions 1 and 3, the ant is permitted to *switch states*, eliminating the need to search for a different cell. Instead, these conditions simulate specific actions: in the first, the ant resets its food reference to *null*, mimicking the deposit of food into the colony, while in the third, it acquires a reference to a food source, reducing the food's interaction counter by 1.

Conditions 2 and 4 represent the ant's general wandering behavior, provided by the `GetNextPoint()` method, which selects the optimal cell to move towards based on the ant's neighboring cells (a detailed explanation of this method is provided below). The key difference between these conditions is that condition 2 also involves the ant depositing a pheromone value, ranging from 0 to 1, based on the Euclidean distance between its food and the colony. Larger pheromone values are deposited closer to the food source, helping to attract other ants towards it.

Before proceeding to the next ant, the algorithm uses the current ant's present and next positions to *dig* into the terrain through the `Dig()` function, which modifies the heightmap based on these positions. However, the process goes beyond simple digging; the excavated value is redistributed to surrounding cells according to fixed percentages, as illustrated in Figure 3.1, which details all possible combinations. Finally, the ant's `CurrentCell` is updated, completing its movement.

As mentioned earlier, the `GetNextPoint()` function is the critical point where an ant makes its movement decision based on its surrounding environment. This decision takes into account both heightmap and pheromone matrices' values, as well as a sense of direction towards the colony when carrying food. An extra random value is also added to better simulate a wandering effect, used primarily when searching for food. Algorithm 3 offers an overview of this method, while Table 3.4 offers a small description of each used variable alongside its corresponding symbol in equations present in this section.

It starts by allocating a significant number of local variables (lines 1-10). Any variable containing the word *portions* (lines 2-5) refers to an array of `float` values that represent the collection of calculated values of a specific type (pheromone, slope, direction or random) for each neighbouring cell (e.g.,

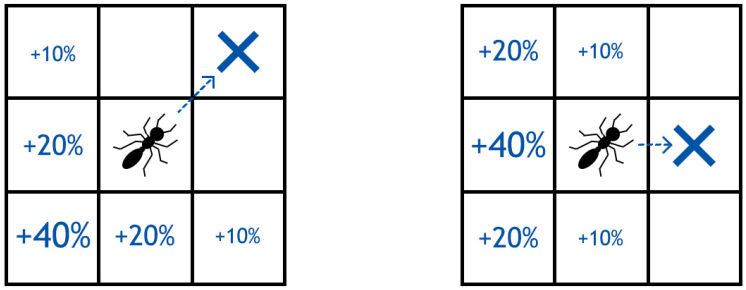


Figure 3.1: Showcase of the terrain redistribution pattern followed by ants in diagonal (left) and horizontal/vertical (right) moves. The ant represents the current cell it's placed on, while the arrow signals the ant's movement towards a new cell (blue cross). The percentages indicate the amount of terrain added to other cells based on the movement.

Symbol	Name	Description
N	`neighboursAmount`	Amount of neighbouring cells.
P	`pheromonePortions[i]`	Calculated pheromone value on a given cell.
S	`slopePortions[i]`	Calculated slope value on a given cell.
D	`directionPortions[i]`	Calculated *direction-to-colony* value on a given cell.
R	`randomPortions[i]`	Calculated random value on a given cell.
W_p	`pheromoneWeight`	Weight of each calculated pheromone value.
W_s	`slopeWeight`	Weight of each calculated slope value.
W_d	`directionWeight`	Weight of each calculated *direction-to-colony* value.
W_r	`randomWeight`	Weight of each calculated random value.
ρ_n	`percentage`	Calculated percentage of each neighbouring cell.
θ	`angle`	Calculated angle between the two direction vectors.

Table 3.4: Calculated values and parameters used in `GetNextPoint()`.

Algorithm 3 GetNextPoint

Input: neighbours, currentCell, destination (optional)
Output: nextCell

```
1: pheromonePortions ← new[]
2: slopePortions ← new[]
3: directionPortions ← new[]
4: randomPortions ← new[]
5: currentHeight ← grid.Heights[currentCell]
6: minHeight ← currentHeight
7: maxHeight ← currentHeight
8: nHeights ← FetchNeighboursHeight(neighbours, ref minHeight, ref maxHeight)
9: totalSum ← 0
10: for i ← 0 to neighbours.Count do
11:     if minHeight = maxHeight then
12:         slopePortions[i] ← 1
13:     else
14:         minDiff ← minHeight - currentHeight
15:         maxDiff ← maxHeight - currentHeight
16:         if absSlope then
17:             minAbs ← |minDiff|
18:             maxAbs ← |maxDiff|
19:             maxAbsDiff ← Max(minAbs, maxAbs)
20:             slopePortions[i] ← CalcSlopePortion(currentHeight, nHeights[i], maxAbsDiff) ·
                    slopeWeight
21:         else
22:             slopePortions[i] ← CalcSlopePortion(currentHeight, nHeights[i], minDiff, maxDiff) ·
                    slopeWeight
23:     if destination is none then
24:         pheromonePortions[i] ← CalcPheromonePortion(grid.Pheromones[neighbours[i]]) ·
                pheromoneWeight
25:         randomPortions[i] ← CalcRandomPortion() · randomWeight
26:     else
27:         mainDirection ← destination - currentCell
28:         direction ← neighbours[i] - currentCell
29:         angle ← AngleBetween(mainDirection, direction)
30:         directionPortions[i] ← antsInPlace and direction = (0,0) ? 0 :
                CalcDirectionPortion(angle) · directionWeight
31:     totalSum ← totalSum + pheromonePortions[i] + slopePortions[i] + directionPortions[i] +
            randomPortions[i]
32: nPerctgs ← new[]
33: for i ← 0 to neighbours.Count do
34:     percentage ← (pheromonePortions[i] + slopePortions[i] + directionPortions[i] + randomPortions[i]) / totalSum
35:     nPerctgs[i] ← (i, percentage)
36: nextCell ← ChooseRandom(neighbours, nPerctgs)
37: return nextCell
```

`pheromonePortions` will store the calculated pheromone values of each neighbour, taking into account the weight selected by the user; this prevents a second calculation of the same values on the second `for` loop).

Also important are the min/max height variables (lines 7-8), which, as the names imply, represent the minimum and maximum height values of the neighbouring cells. They both start with the height value of the cell where the ant is placed, getting updated right after (line 9). By using the keyword `ref` when passing them as arguments to `FetchNeighboursHeight()`, the function is able to treat them as references to the variables and thus update them. This function also returns the collection of heights for each neighbour, which is stored in `nHeights`.

The method then enters one of its two loops (both simply iterate over the number of neighbours, represented by the `neighboursAmount` variable). This first loop is responsible for calculating the values of each *navigation cue* (pheromone concentration, slope differences, etc.), which range from 0 to 1. It starts by verifying if the previously calculated values for `minHeight` and `maxHeight` are equal. If true, this means the ant's surrounding terrain is flat, and thus, no slope calculations need to be made, instead attributing a similar *slope* value to all neighbours (line 13). If false, two different techniques can be used to calculate said value. Both start by calculating the differences between the found `minHeight` and `maxHeight` with the cell's `currentHeight` (lines 15-16). The subsequent check (line 17) verifies which technique the user will choose.

The `absSlope` approach uses absolute values of some previously calculated variables (lines 17-21). First, it converts `minDiff` and `maxDiff` to their correspondant absolute values (lines 18-19). Then, a simple check is made to determine which value is bigger, with it being saved on a new variable `maxAbsDiff` (line 20). Finally, the *slope portion* value is calculated for the current neighbour through the use of the `CalcSlopePortion()` function, represented as the first calculation in Equation 3.1. The result of using this approach is that, as all cells have a positive value, the ant gets to choose those with less slope.

$$S = \begin{cases} 1 - \frac{|to-from|}{maxAbsDiff}, & \text{if } absSlope = \text{true} \\ 1 - \frac{to-from+|minDiff|}{|minDiff|+|maxDiff|}, & \text{otherwise.} \end{cases} \tag{3.1}$$

The second approach (lines 22-23) also uses the absolute values of some previously mentioned variables, with the differences in the *overload* of `CalcSlopePortion()` (the second calculation in Equation 3.1) resulting in the ant favouring downward slopes.

The next `if` statement (lines 24-31) verifies whether a destination's location was passed into the method, differentiating a wandering ant from those that carry food and wish to return to the colony. If no destination was given (lines 24-26), then the ant is wandering and has interest in one of the main *navigation cues*: pheromones. The `CalcPheromonePortion()` method simply returns the amount of pheromone in a given location by fetching it from its respective matrix (any pheromone amount is clamped between 0 and 1 when being deposited, thus not requiring any further calculations in this step). An extra value also gets calculated (line 26) to aid create a more random, less deterministic cell selection.

On the other hand, if a destination was given (i.e., the ant is carrying food and wants to return to

the colony), then a *direction-to-colony cue* is calculated to influence the ant towards it (lines 27-31). By determining the angle θ between two main vectors (current-to-destination and current-to-neighbour), a value from 0 to 1 can be obtained by using the formula in Equation 3.2 (note that the vectors are two-dimensional, with height values not taken into account).

$$D = 1 - \frac{\theta}{180} \tag{3.2}$$

Finally, at the end of the first loop, the `totalSum` variable is incremented with any previously calculated values, by summing up all *portion* arrays.

The second loop (lines 34-36), as explained previously, goes through the same amount of iterations as the previous one. This loop's responsability is to calculate the percentage of choosing each neighbour, by dividing the sum of its previously determined *navigation cue* values by the total amount of these same values across the entire neighbours array (line 35). This percentage is stored in a tuple array (`nPerctgs`) along with the neighbour's index (which corresponds to the loop's index). The mathematical formula for this operation is represented in Equation 3.3.

$$\rho_n = \frac{W_p \cdot P + W_s \cdot S + W_d \cdot D + W_r \cdot R}{\sum_{i=1}^{N} W_p \cdot P + W_s \cdot S + W_d \cdot D + W_r \cdot R} \tag{3.3}$$

With all the previously obtained information, all that's left to do is determine which neighbouring cell to choose. The function `ChooseRandom()` offers a standard implementation of a roulette wheel selection, which returns a random member of the supplied collection (in this case, `neighbours`) according to the percentage of each of its elements. This value is then returned by the `GetNextPoint()` method, consequently reaching its end.

3.1.3 Pheromone Diffusion and Evaporation

Updating the pheromone matrix is the final step regarding the main algorithm. This is done by applying two techniques that mimick how real pheromones would get dispersed: diffusion and evaporation. Algorithm 4 displays this process, while Table 3.5 showcases equation symbols and descriptions of used variables.

Algorithm 4 UpdatePheromones

```
1:  newPheromones ← new[,]
2:  for y ← 1 to grid.Dimension - 1 do
3:      for x ← 1 to grid.Dimension - 1 do
4:          currentPhCon ← grid.Pheromones[y,x]
5:          neighbrsTotalPhCon ← 0
6:          for k ← 1 to 8 do
7:              neighbrsTotalPhCon ←
                    neighbrsTotalPhCon + grid.Pheromones[y + neighboursY[k], x + neighboursX[k]]
8:          if neighbrsTotalPhCon ≠ 0 or currentPhCon ≠ 0 then
9:              newPheromones[y, x] ← (1 - phEvap) ·
                    (currentPhCon + (phDiff · ((neighbrsTotalPhCon / 8) - currentPhCon)))
10: grid.Pheromones ← newPheromones
```

Symbol	Name	Description
N_{ph}	`grid.Pheromones[neighbour]`	Pheromone concentration value on a cell's neighbour.
C_{ph}	`currentPhCon`	Pheromone concentration value on a cell.
W_e	`phEvap`	Weight linked to pheromone evaporation.
W_d	`phDiff`	Weight linked to pheromone diffusion.
U_{ph}	`newPheromones[y,x]`	Updated pheromone value.

Table 3.5: Calculated values and parameters used in `UpdatePheromones()`.

The function `UpdatePheromones()` starts by creating a new pheromone matrix (line 1). This allows any future calculations and subsequent storing of said values to be made independently from the main pheromone matrix (a method sometimes called *double buffering*). Right after, two nested `for` loops take place in order to traverse the entire matrix space. However, by starting both `x` and `y` variables with a value of `1`, and subtracting `1` from the grid's `Dimension`, the outer layer is ignored in order to save in performance by conjugating this with pre-calculated *neighbour-finding* arrays (used in line 7) that remove the extra overhead of having to calculate the directions of each cell's neighbour. This also removes the necessity of checking if a neighbour is within bounds.

In lines 4 and 5 two local variables are created. The first one is simply storing the amount of pheromone in the currently evaluated position. The second one represents the total amount of pheromones spread throughout all found neighbours. It gets its value incremented in the subsequent `for` loop (lines 6-7), which iterates through all 8 neighbours (following the structure of a Moore neighbourhood). After this loop, pheromone diffusion and evaporation occurs within a single instruction by using the formula in Equation 3.4 (however, if the current cell or its neighbours possess no pheromone concentration, the cell is skipped).

$$U_{ph} = (1 - W_e) \cdot (C_{ph} + W_d \cdot (\frac{\sum_{i=1}^{8} N_{ph}}{N} - C_{ph})) \tag{3.4}$$

The evaporation section of the formula is responsible for removing a percentage of the current pheromone amount on a given cell, controlled by the user through the `phEvap` parameter. On the other hand, diffusion allows a cell to integrate into itself a portion of the pheromone amount on its neighbours, while also taking into account its own concentration level, and can be controlled with the `phDiff` parameter. Figure 3.2 offers an example of pheromone concentration at the end of a simulation cycle.

Finally, before exiting this function, the local pheromone matrix is assigned to the main matrix, effectively updating and replacing its values.

3.2 Ant Colony Optimization Solution

During the early stages of this project, an alternative approach was explored for agent-based landscape surface exploration, based on the previously discussed ACO algorithm (Algorithm 5). This method adapted ACO's graph-based structure to a three-dimensional terrain using custom techniques, enabling

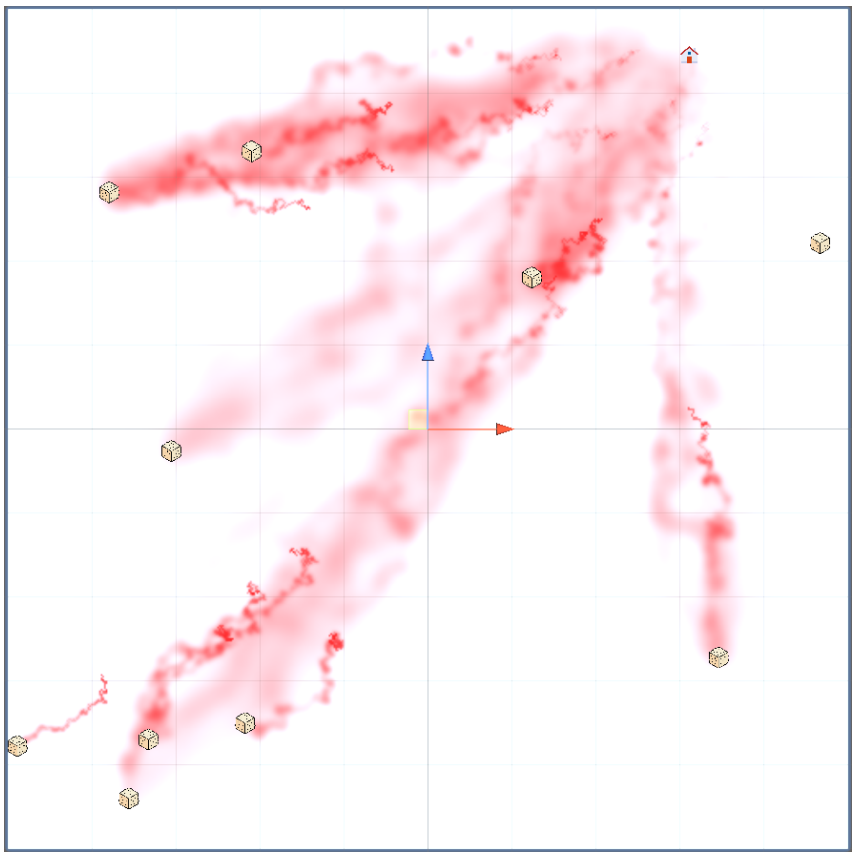


Figure 3.2: Image depicting pheromone concentration on the terrain with heightmap display disabled. *Sugar cubes* represent food across the landscape with the *home* icon representing the ants' colony. A white-to-red gradient is used to showcase the respective low-to-high pheromone concentration.

ants' trails to directly impact landscape modifications. However, the algorithm faced significant limitations in enabling proper terrain exploration, as ant movement was constrained by the predefined locations of graph nodes. Ants could only 'jump' from node to node, relying primarily on connection-based pheromones as their navigation cues. Lacking the ability to fully assess and interact with their environment, this approach proved inadequate. As a result, the *Landscaper Ants* algorithm was developed, allowing agents to make decisions based on features present in the terrain itself and serving as the primary focus of this project.

Algorithm 5 AntColonyOptimization

```
1: ants.Create()
2: graph.Generate()
3: while iterations not over do
4:     for each ant do
5:         graph.CalculateTrail(ant)
6:     graph.UpdatePheromoneTrails()
7:     GetBestTrail()
```

The results from this initial exploration can be found in Section 4.1.

3.3 Unity Implementation

This section highlights the Unity implementation of the previously discussed *Landscaper Ants* algorithm, which serves as this project's primary focus. It begins with an introduction to the project's structure, followed by an overview of the categorization of its parameters. The final part introduces the concept of experiments and explains how users can generate results in bulk, enabling more refined outcomes in future uses of this algorithm.

3.3.1 Software Architecture

The project's main components are represented in Fig. 3.3. The class at the top of the image is the Unity game engine-provided [36] `MonoBehaviour` class, which is the required base class for all Unity scripts attached to game objects[1].

Core functionality is contained within the `LandscapeController` class. It is here where the algorithm's parameters can be set, and the processing of the ants' behaviour occurs. To accomplish that, it makes use of two other classes, namely:

- The `Ant` class, which holds information about individual ants, namely their grid position and some helper functions such as the ones that dictate their current state (e.g., looking for food).
- The `Grid` class, which contains the representation of some important structures, namely the height and pheromone grids and collection of `Food` present within the given limits, and their corresponding creation methods.

[1] A `GameObject` represents everything in a Unity project, including characters, props, and scenery.

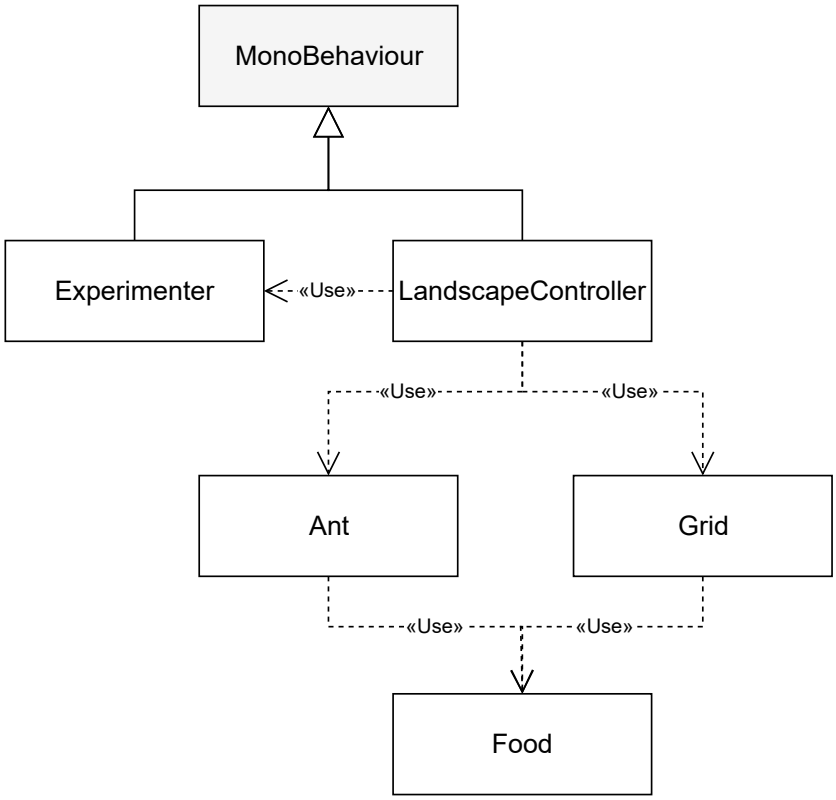


Figure 3.3: Simple UML class diagram representing the project's structure. MonoBehaviour is a Unity class.

Lastly, the `Experimenter` class serves as a container for parameters' values which users might want to test in an automatic, iterative fashion. The next subsection provides more details about this class's parameters and functionality.

3.3.2 Experiments

Much like ACO and other search-based approaches, the *Landscaper Ants* algorithm requires careful parameter fine-tuning to identify values best suited to the specific problem at hand. However, as the number of parameters increased, manual tuning became impractical due to the vast range of potential combinations, particularly when considering the mix of float and integer values. This theoretical infinity of possibilities necessitated a solution to streamline the process and accelerate the generation of visual results. To address this challenge, a dedicated class was developed to facilitate the creation of 'experiments' that test specific parameter combinations defined by the user.

The `Experimenter` class enhances productivity by allowing users to store multiple parameter combinations and execute the algorithm with each combination sequentially, all in one automated process. This approach enables users to 'mix and match' parameters without needing to manually start each run, wait for it to finish, and then initiate the next. Instead, the system handles all executions in a streamlined manner. For each run, the class generates visual results in image format and records the parameter values used in a text file, ensuring reproducibility. Users can configure these settings within a dedicated

`GameObject`, where parameters are intuitively categorized into two primary groups for ease of setup.

Experiment Parameters This category contains collections of values to be assigned to parameters that directly influence an algorithm's run and subsequent results. This includes *seeds* to be supplied to Unity's random number generator, the quantity of ants to be created, the amount of food to be generated, among others. Any collection can have as many elements as the user desires, although, for collections linked to `boolean` values, it is prudent to have 1 to 2 values, as these parameters can only be either true or false.

Camera Screenshot Parameters This section possesses two collections of values related to position and rotation of the camera responsible for print screens (in order to obtain visual results from different perspectives). Also present is `printStep`, a variable that controls at which step to take said print screens (e.g., a value of 500 translates to screenshots being taken every 500^{th} step of each experiment run).

The `RunExperiments()` function, responsible for executing experiments, operates within a coroutine. This approach allows the function to run alongside Unity's main thread, enabling real-time updates to the Editor's display and the creation of terrain images reflecting the current state. The function organizes the main algorithm within a series of nested `for` loops, each iterating over specific parameter collections. Once the parameters for a particular experiment are set, the system creates directories to save the generated images.

Upon the main algorithm's execution a text file is generated containing the relevant parameter values used in the experiment. This file ensures reproducibility by allowing users to replicate the results manually if needed. Additionally, a fail-safe mechanism is in place to skip previously completed experiments by checking for the existence of the corresponding text file. This avoids redundant computations, which is particularly important in cases where external events, such as power outages, interrupt the experiment process—especially since some experiments may take several days to complete.

By using the aforementioned method—whose results are exposed and thoroughly discussed in the next chapters—over 4000 consecutive experiments were obtained over a period of 14 days on a computer containing an AMD Ryzen 16-core CPU, 64GB of DRAM and an Nvidia RTX 3070 GPU running Ubuntu 22.04 LTS. An extra 600 experiments (which contained manually tuned parameter values) were acquired by running an additional set for around 4 days on a computer containing an Intel Core i7-8750H CPU, 16GB of DRAM and an Nvidia GTX 1070 GPU running Windows 10.

Chapter 4

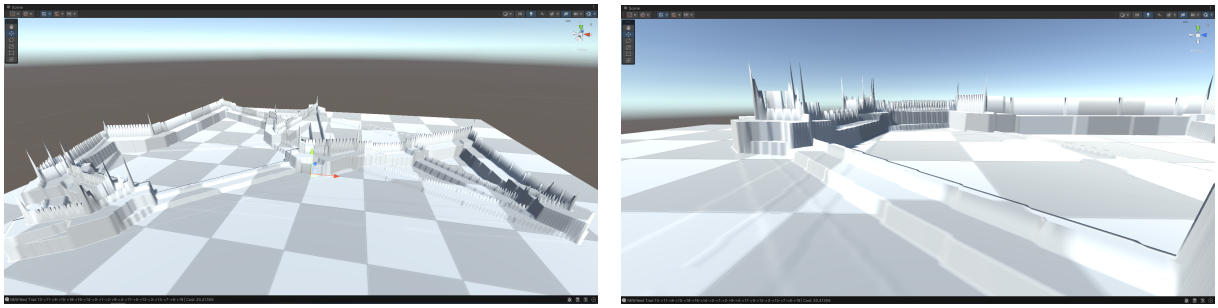
Results

This chapter presents the outcomes of the research, focusing on landscape modification achieved through agents exhibiting ant-like behaviors. The first section provides a brief overview of the results obtained from the ACO approach detailed in Section 3.2. The second section highlights selected experiments from the *Landscaper Ants* algorithm, showcasing intriguing parameter combinations that produce compelling visual results. The chapter concludes with a brief exploration of a preliminary implementation of the algorithm using custom landscapes.

4.1 ACO Castles

Figure 4.1 showcases two different perspectives on the same interesting castle-like structure obtained from the ACO-based solution. The elevated terrain seen in both images corresponds to the projection of ants' trail patterns after a full run of the algorithm. Users had control over both the elevation value and the width of the projected trails, enabling ants to modify the landscape within a specific radius, which explains the uneven increases in elevation, particularly in cells representing the graph's nodes and those directly connecting them.

As discussed in previous chapters, while the results showed potential for generating artificial structures, this solution was ultimately abandoned in favor of a more robust, exploration-based approach.



(a) Wide View (b) Close-up

Figure 4.1: Castle-like shapes created by terrain deformation through the use of ACO.

4.2 Landscaper Ants - Experiments

The following subsections present two sets of experiments, each with a clearly defined purpose. The results of selected experiments are analyzed in detail, along with explanations of their corresponding parameters, ensuring a clear understanding and facilitating future reproducibility if needed.

4.2.1 Experiments - 1st Set

The first set's main objective was to determine which parameters mostly affected the overall behaviour of ants and their *landscaping* capabilities. As such, most parameters needed multiple, diverse values in order to cover a large amount of possible outcomes. The total amount of parameter combinations ended up surpassing the one million mark, although reaching that amount was not necessary (or predictable) as constantly checking experiments' results and corresponding parameter values offered a good overview of which values generated better results.

Experiment 4

Table 4.1 presents the used parameter values for this experiment. Here, the focus was to test an environment where few ants exist opposite to a higher amount of food. All ants start at random positions and can take a maximum of 10 000 steps (which can include steps back to the same position, given the value of `antsInPlace`). Weight values are evenly distributed, reducing any pre-calculated value to a quarter of itself (i.e., the calculated pheromone value of a specific cell gets reduced by 75%). Pheromone evaporation and diffusion values are both set to 0.2, meaning that any cell that contains some pheromone value will lose 20% of itself as well as spread another 20% to its neighbours. Lastly, ants are to explore and modify a flat plane, digging much further when possessing food (which, in this instance, carves better trails). Figure 4.2 showcases the progression of obtained visual results.

Experiment 2989

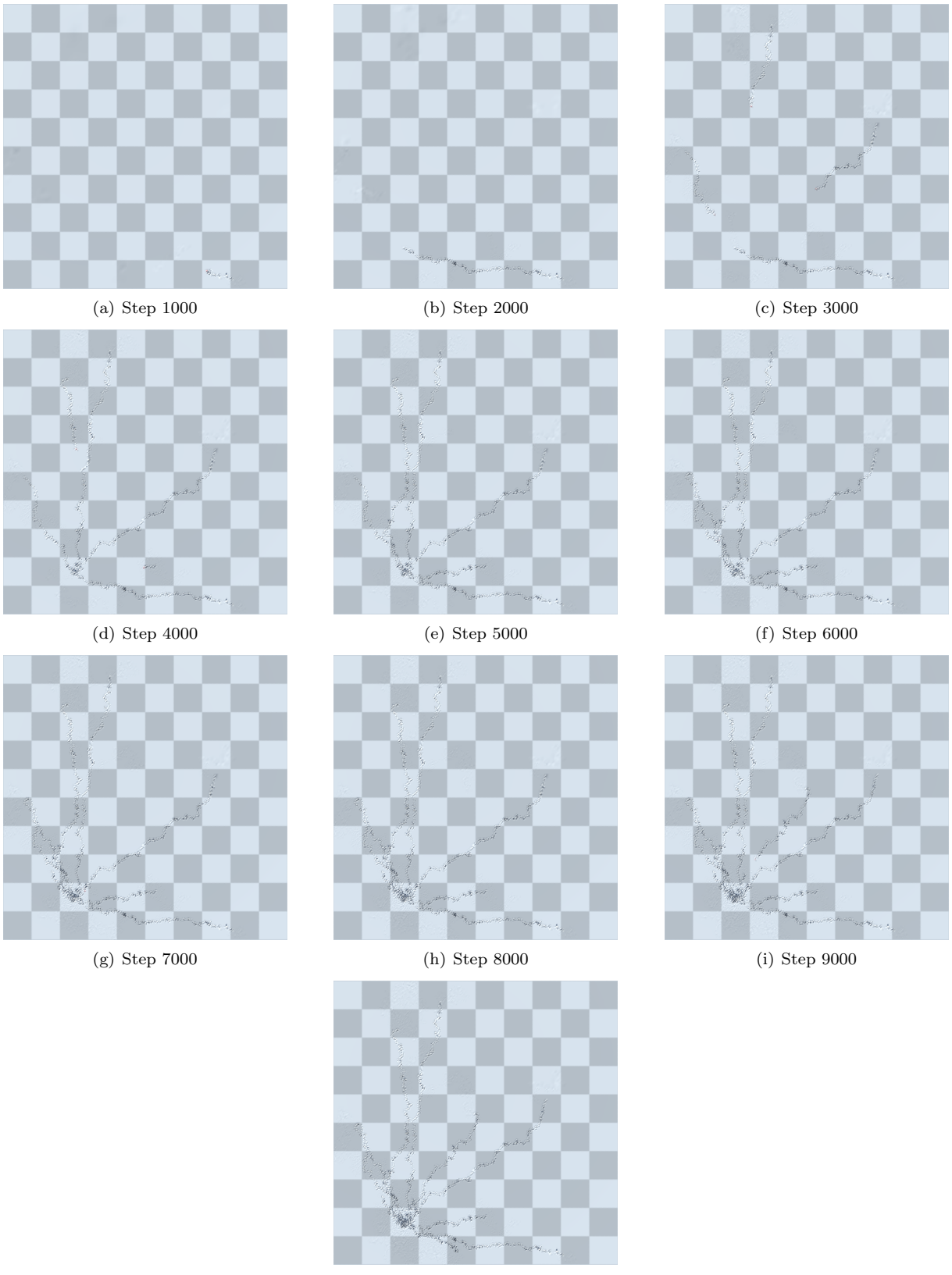
This experiment's parameter values can be found in Table 4.2. Much like the previous experiment, it also possesses a higher amount of food than that of ants. Ants also start at random locations, rather than their colony and take the same amount of steps (as well as being able to stay in the same cell and use the same slope verification method). The major difference is the values given to the different available weights. In this instance, the biggest value is given to `randomWeight` (60%), while all others get a much smaller one (10%). This means that randomness plays a big part in ants' behaviours in this particular experiment. Pheromone evaporation and diffusion values have increased (both set to 60%) and so has the amount of excavated terrain whilst ants carry food (wandering ants have had this value decreased). Finally, the initial terrain is formed with Perlin Noise, which also contrasts with the previous experiment. The combination of these parameter values lead to the results found in Figure 4.3.

Parameter Name	Value
rndSeed	1726737503
shuffleAnts	true
individualStart	true
nAnts	10
maxSteps	10000
antsInPlace	true
absSlope	true
pheromoneWeight	0.25
slopeWeight	0.25
directionWeight	0.25
randomWeight	0.25
phEvap	0.2
phDiff	0.2
foodHeightIncr	0.05
noFoodHeightIncr	0.002
flatTerrain	true
baseDim	513
foodAmount	100
maxFoodBites	∞

Table 4.1: Assigned parameter values in experiment 4.

Parameter Name	Value
rndSeed	1726737503
shuffleAnts	true
individualStart	true
nAnts	10
maxSteps	10000
antsInPlace	true
absSlope	true
pheromoneWeight	0.1
slopeWeight	0.1
directionWeight	0.1
randomWeight	0.6
phEvap	0.6
phDiff	0.6
foodHeightIncr	0.02
noFoodHeightIncr	0.01
flatTerrain	false
baseDim	513
foodAmount	100
maxFoodBites	∞

Table 4.2: Assigned parameter values in experiment 2989.



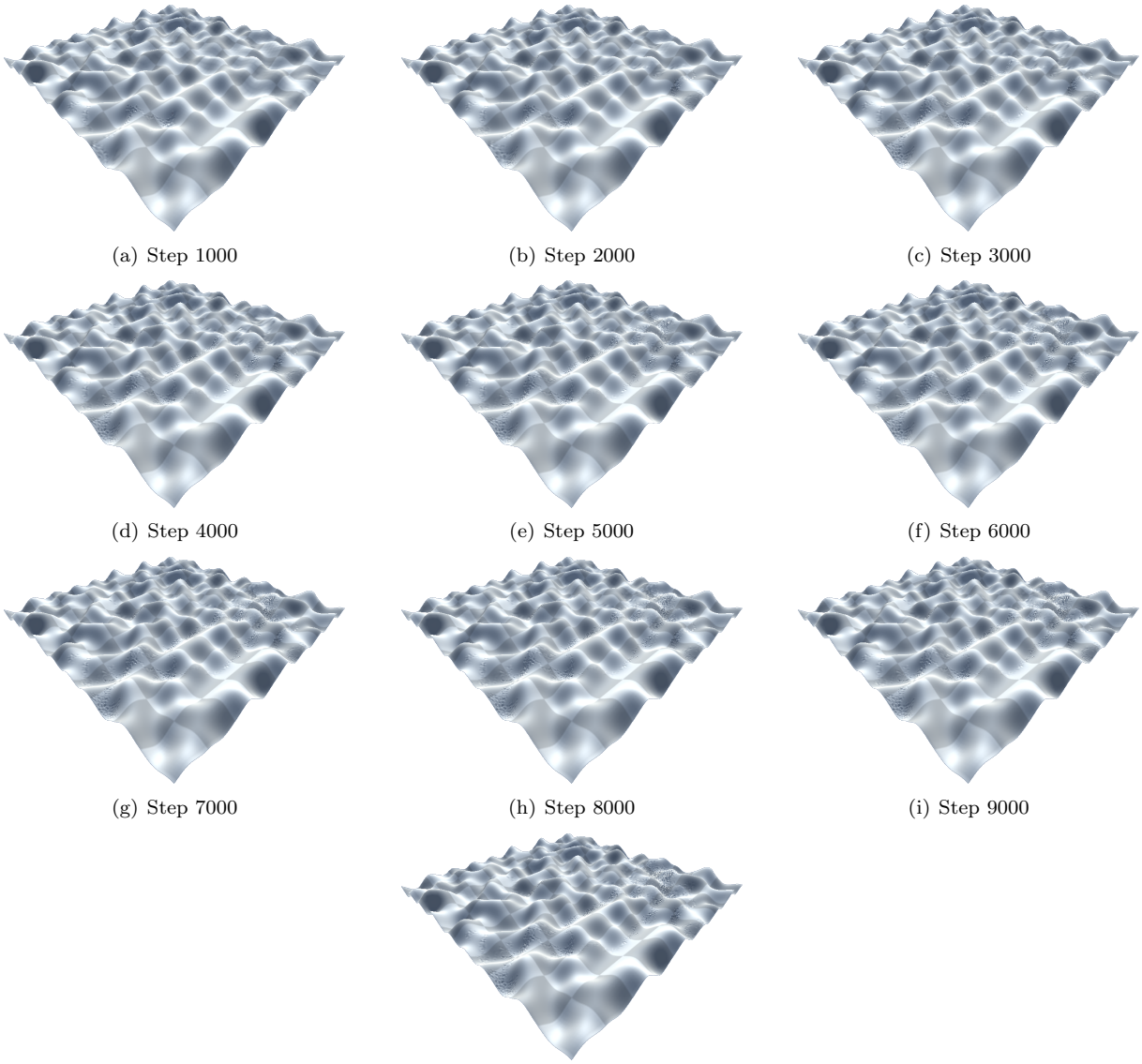
(a) Step 1000 (b) Step 2000 (c) Step 3000

(d) Step 4000 (e) Step 5000 (f) Step 6000

(g) Step 7000 (h) Step 8000 (i) Step 9000

(j) Step 10 000

Figure 4.2: Experiment 4's progressive visualization derived from ant movements after 10 000 iterative steps.



(a) Step 1000 (b) Step 2000 (c) Step 3000

(d) Step 4000 (e) Step 5000 (f) Step 6000

(g) Step 7000 (h) Step 8000 (i) Step 9000

(j) Step 10 000

Figure 4.3: Experiment 2989's progressive visualization derived from ant movements after 10 000 iterative steps.

Parameter Name	Value
rndSeed	1726737503
shuffleAnts	true
individualStart	true
nAnts	10
maxSteps	10000
antsInPlace	true
absSlope	false
pheromoneWeight	0.6
slopeWeight	0.1
directionWeight	0.1
randomWeight	0.1
phEvap	0.4
phDiff	0.8
foodHeightIncr	0.06
noFoodHeightIncr	0.03
flatTerrain	true
baseDim	513
foodAmount	100
maxFoodBites	∞

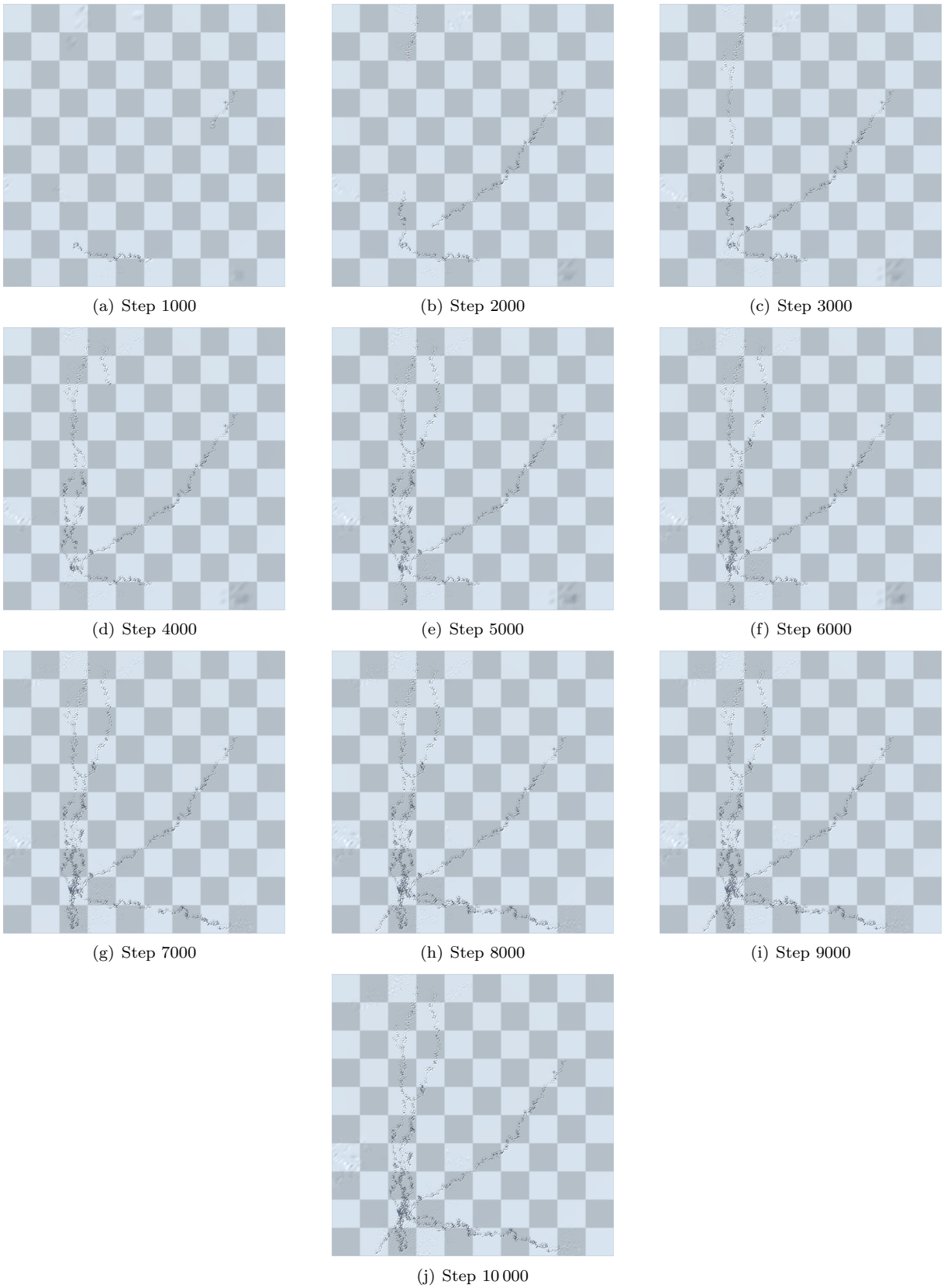
Table 4.3: Assigned parameter values in experiment 4134.

Experiment 4134

Table 4.3 contains the parameter values used in this experiment. Just like the previous experiments, it continues to test low amount of ants versus high amount of food, just as they continue spawning in individual cells. The major differences here come from the fact that the *absolute slope technique* is not in use (meaning ants now prefer downward slopes) and that there's a bigger value attributed to `pheromoneWeight` (60%) when compared to other weights (10%). The parameters for evaporation and diffusion also had their values changed, notably `phDiff`, which almost *dissipates* the entire amount of pheromone present on a cell (by distributing to others). Digging suffers an enormous increase on both ant behaviours, meaning ants will generally dig more than in previous experiments. Also of note is the reappearance of a flat terrain. Figure 4.4 presents the obtained results.

4.2.2 Experiments - 2nd Set

This set's main goal was to tune parameter selection based on previous experiments and create collections of parameters that were sure to produce better results. This, of course, also required a thorough analysis, not only to confirm that the selected values generate good results, but also further tune them.



(a) Step 1000 (b) Step 2000 (c) Step 3000

(d) Step 4000 (e) Step 5000 (f) Step 6000

(g) Step 7000 (h) Step 8000 (i) Step 9000

(j) Step 10 000

Figure 4.4: Experiment 4134's progressive visualization derived from ant movements after 10 000 iterative steps.

Parameter Name	Value
rndSeed	1726737503
shuffleAnts	true
individualStart	true
nAnts	1000
maxSteps	10000
antsInPlace	false
absSlope	true
pheromoneWeight	0.7
slopeWeight	0.1
directionWeight	0.1
randomWeight	0.1
phEvap	0.05
phDiff	0.05
foodHeightIncr	0.04
noFoodHeightIncr	0.002
flatTerrain	false
baseDim	513
foodAmount	1
maxFoodBites	∞

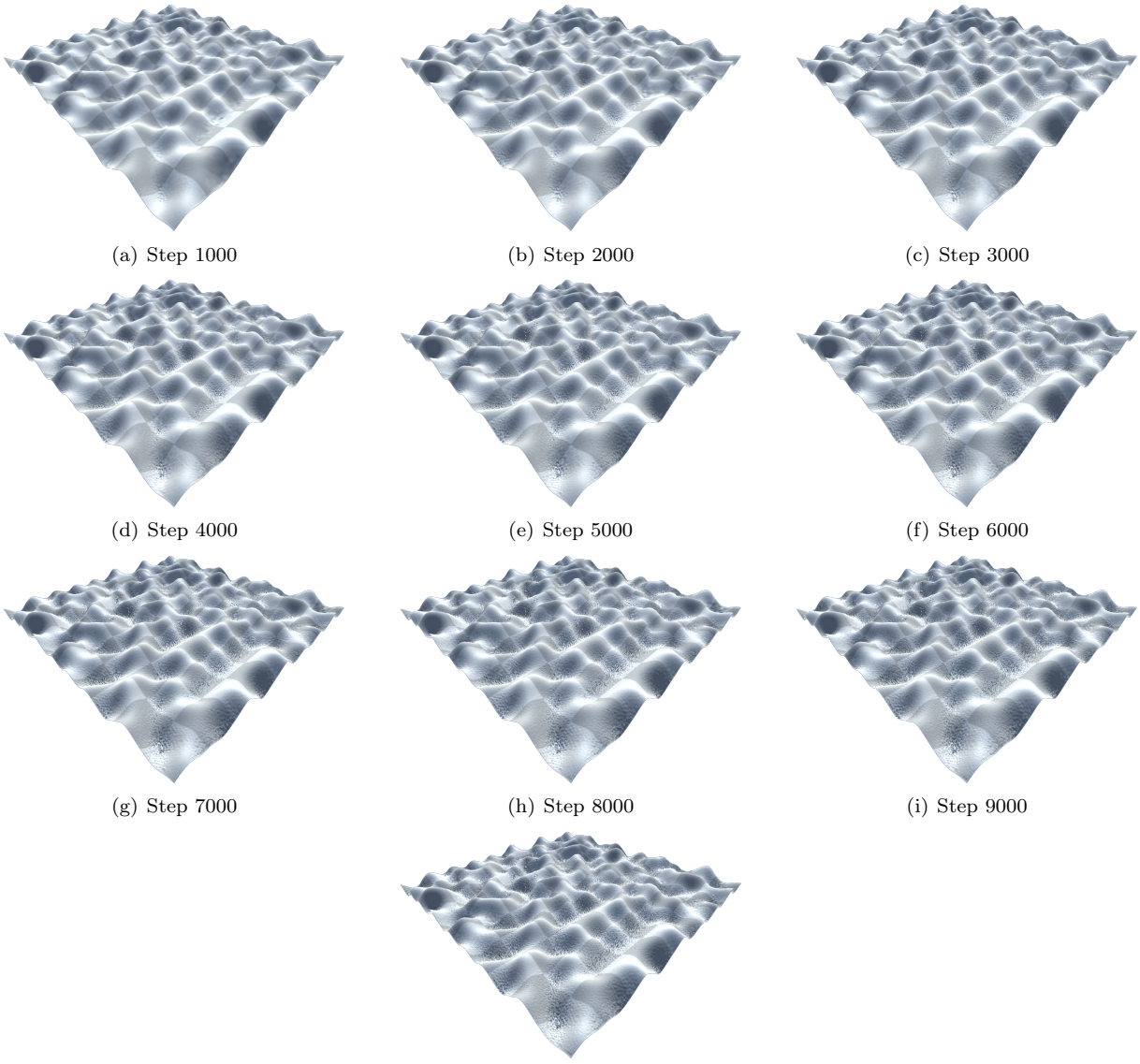
Table 4.4: Assigned parameter values in experiment 16.

Experiment 16

This experiment's assigned parameter values are exposed in Table 4.4. Contrary to other experiments from the previous set, a high number of ants is this time tested opposite to a low amount of food. Ants can also no longer stay in place, forcing them to move to a different cell after each algorithmic step. Weights have been readjusted, this time granting a much higher weight value to pheromone calculations (70%), when compared to other values of the same *type* (10%). Evaporation and diffusion values have also been modified, both possessing an extremely small value of only 5%. Lastly, both digging values remain fairly low (with a higher amount applied when ants carry food) which combines well with the non-flat Perlin-based terrain. Visual results can be found in Figure 4.5.

Experiment 615

Table 4.5 exposes the parameter values assigned in this experiment, this time testing the same amount of ants as in the previous one against a higher amount of food. The variable responsible for the weight given to slope calculations also has a bigger value (70%), with `pheromoneWeight` having its value reduced to match other weights (10%). Evaporation and diffusion values have their values slightly decreased and increased, respectively. Finally, both digging values are reduced to extremely low (close) values, which, again, work well with the non-flat terrain. Figure 4.6 showcases the visual progression obtained from the previous parameter values.



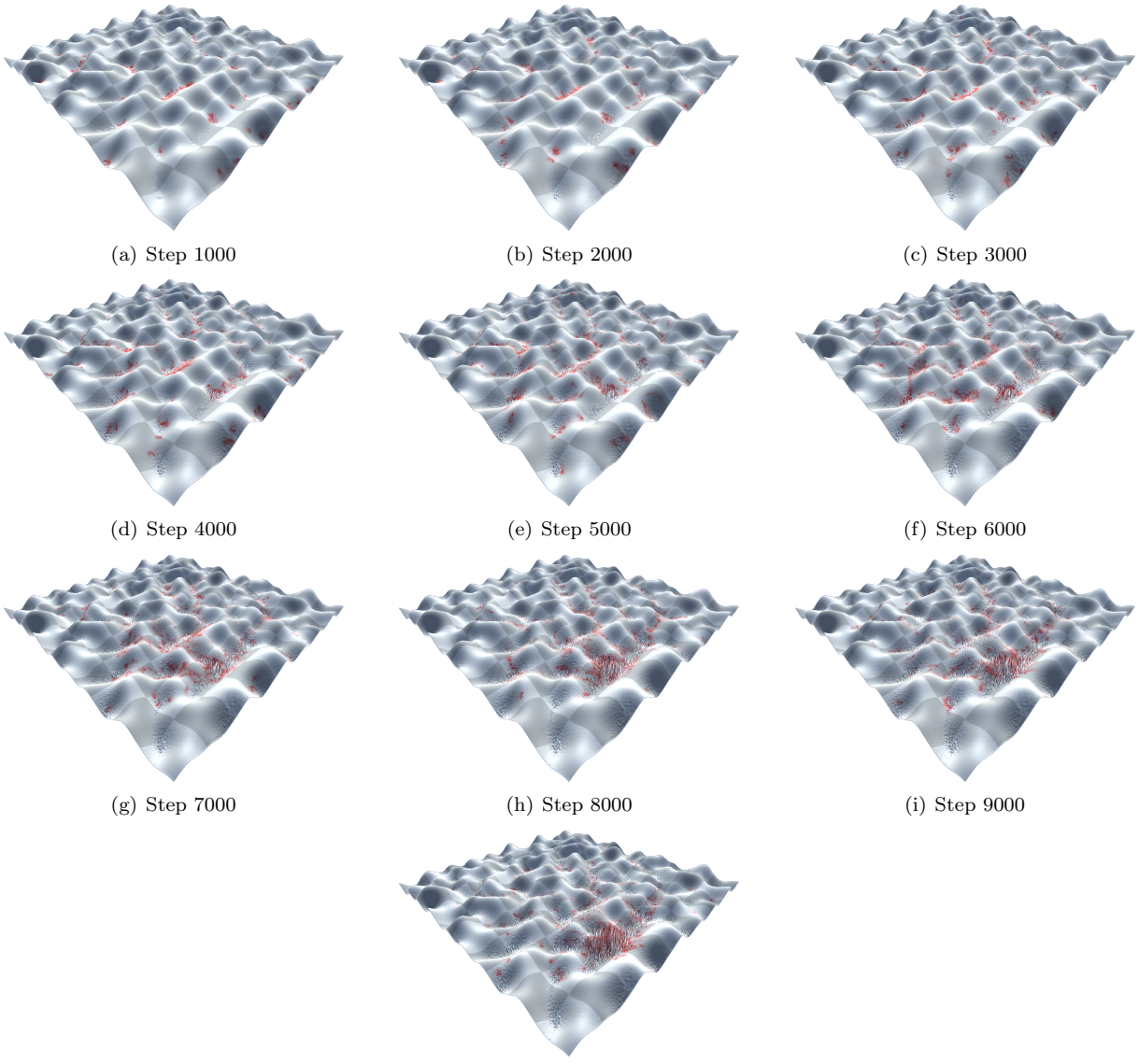
(a) Step 1000 (b) Step 2000 (c) Step 3000

(d) Step 4000 (e) Step 5000 (f) Step 6000

(g) Step 7000 (h) Step 8000 (i) Step 9000

(j) Step 10 000

Figure 4.5: Experiment 16's progressive visualization derived from ant movements after 10 000 iterative steps.



(a) Step 1000 (b) Step 2000 (c) Step 3000

(d) Step 4000 (e) Step 5000 (f) Step 6000

(g) Step 7000 (h) Step 8000 (i) Step 9000

(j) Step 10 000

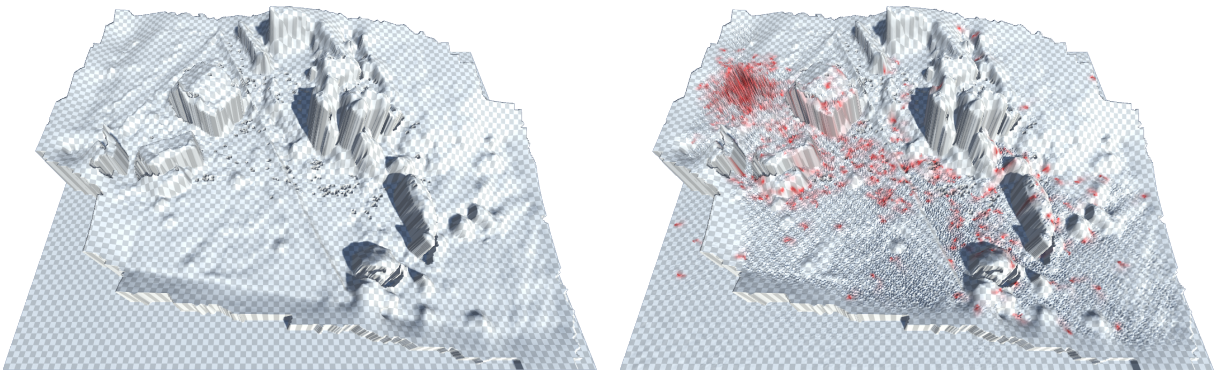
Figure 4.6: Experiment 615's progressive visualization derived from ant movements after 10 000 iterative steps.

Parameter Name	Value
rndSeed	-2059401394
shuffleAnts	true
individualStart	true
nAnts	1000
maxSteps	10000
antsInPlace	false
absSlope	true
pheromoneWeight	0.1
slopeWeight	0.7
directionWeight	0.1
randomWeight	0.1
phEvap	0.02
phDiff	0.1
foodHeightIncr	0.002
noFoodHeightIncr	0.001
flatTerrain	false
baseDim	513
foodAmount	100
maxFoodBites	∞

Table 4.5: Assigned parameter values in experiment 615.

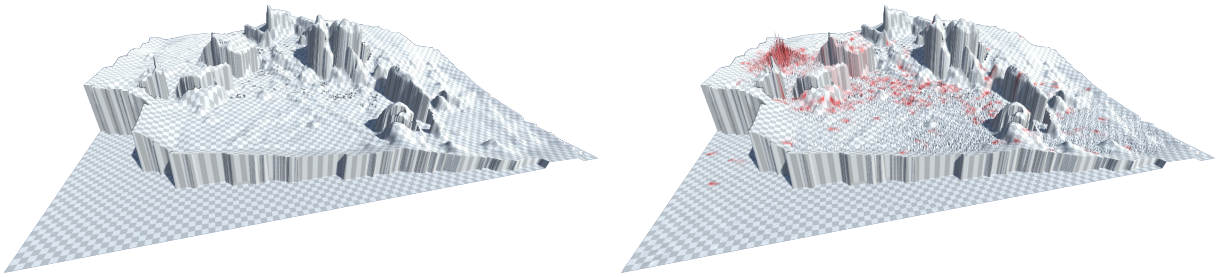
4.2.3 Preliminary Results with Custom Landscapes

Due to the ants' navigational and landscaping capabilities in complex environments, preliminary tests were also executed on a real life-based terrain obtained through drone mapping software. Results were acquired through the use of the same parameters as in Experiment 615, although excavation values were slightly adjusted to improve clarity in the visualization. Figures 4.7 and 4.8 present the original terrain and subsequent transformation after running the main algorithm.



(a) Before (b) After

Figure 4.7: Top-down view of the algorithm's usage on a real life-based custom landscape.



(a) Before

(b) After

Figure 4.8: Side view of the algorithm's usage on a real life-based custom landscape.

Chapter 5

Discussion

This chapter provides an in-depth analysis of the previously described experiments, followed by a discussion of the core implementation ideas behind the algorithm and their impact on the results. Finally, it examines the application's limitations and suggests basic solutions.

5.1 Experiments' Analysis

As showcased, experiments were separated into 2 main sets with complementary testing of the algorithm on a custom (real-life based) terrain. These sets each had a specific purpose regarding manual fine-tuning of behaviour-controlling parameters (such as the weights related to *navigation cues*) through collections of to-be-assigned values stored in a Scene-present `GameObject`.

As the first step was to understand the consequences of the different assignable parameters, the collections created for the first set of experiments contained a high amount of parameter combinations, covering a large amount of possible outcomes. Experiments from this set, along with those from all other sets, were selected based on their visual outcomes.

In Experiment 4, the trails left by ants whilst carrying food were the most visible, while wandering ants produced mostly no terrain alterations. This difference seems to be easily explained by the values attributed to both digging parameters, although there are other factors at play that help describe why trails look so defined. The first is the higher amount of food sources (100) in contrast to a lower number of ants (10), whilst the second is the presence of a flat terrain mixed with the `absSlope` calculation method, which helps ants look for levelled out terrain instead of downward slopes. Hence, when carrying food, the surrounding terrain does not create a major challenge, allowing ants to return back to the colony in an almost linear path. Weights all share the same value, which translates to no *navigation cue* having a larger influence over others.

Experiment 2989's results mostly showcase the same type of visual progression as the last experiment, only now ants are placed on an already altered terrain (by applying Perlin noise on the heightmap before the algorithm has begun). Most other values remain the same, with the major differences in ant behaviour coming from the increased weight value given to the randomness of choosing an ant's

next location. Evaporation and diffusion values were also increased (which is evidenced by the lack of red *splatter* patterns on the terrain), which reduces the likelihood of ants influencing each others paths (when searching for food, since pheromone values have no influence on an ant returning to the colony).

Experiment 4134 displays similarities to Experiment 4 in regards to the type of initially generated flat terrain, although the new parameter values offer the discovery of other food locations as well as different routes, thus producing different trails (a reminder that the same `rndSeed` has been supplied to Unity's `Random` class in these experiments, consequently providing ants with the same colony and initial locations as well as food source locations). This time, ants are much more *aggressive* towards finding pheromones left by others, increasing the chances of finding food. The `absSlope` technique is not used here, with ants preferring downward slopes (which might explain why they seem to follow other's trails - when entering a carved path made by another ant they prefer that same path, since deviating would mean returning to a bigger height level when compared to where they stand).

Some light conclusions can be drawn from this first set of experiments:

1. Low amount of ants in combination with a high amount of food leads to low coverage of the terrain.

2. Currently tested evaporation and diffusion values seem too high, eliminating any pheromone presence after a small amount of steps.

3. Bigger influence of pheromones' weight when choosing an ant's next cell can more easily lead them towards already discovered food sources.

These conclusions were used as starting points for the next experimental set.

Firstly, future experiments were to use much larger numbers of ants while keeping food sources to a minimum. This contrasts with what had been tested thus far and should result in an increased coverage of the terrain, hopefully creating more robust, interesting landscapes. Secondly, pheromone evaporation and diffusion values were to be reduced substantially in order to more easily allow ants to follow pheromone trails created by others and consequently increase the chances of wandering ants finding food. Lastly, pheromone and slope weights were to be given higher priority/values over direction and randomness. Since the biggest percentage of ants will be looking for food, this should allow them to better traverse the landscape.

Experiment 16 is the first of the two chosen experiments in the second set that follow the previous indications. The used seed is the same as in previous experiments, although the drastic change in the amount of ants, the weight of pheromones and low evaporation and diffusion values result in a considerable difference in ant behaviours and thus showcase a much bigger coverage of the entire terrain. Also, for the first time, the parameter `antsInPlace` was set to `false`, forcing ants to move to a different location every time they were to choose one, allowing them to better disperse throughout the landscape. The progression present in the images in the previous chapter also displays ants grouping in the *valleys* created by the Perlin-noise algorithm (even including those that spawned in the upper areas), which could pose a problem if food sources are in places they can no longer reach (since once they move down, they seem

to no longer be able to move upwards, unless they detect pheromones generated by other ants; increasing food sources to a slightly higher amount could help mitigate this).

Experiment 615 also introduces some not yet seen landscape modifications, namely from the use of a different `rndSeed` and further tuned, small evaporation and diffusion values (which help pheromones linger on the terrain for a greater amount of time as evidenced by the red splatters present on the ants' trails). Interestingly, when compared with the previous experiment, this one's visual results appear to be much less chaotic, with ants seemingly finding their way back to the colony much faster and not navigating aimlessly as much, which is explained by increase in the number of food sources spread throughout the landscape. The increase in the weight value controlling traversal decisions regarding slopes has also certainly contribuited to the non-accumulation of ants in terrain depressions.

Some new conclusions can be determined from this second set:

1. Larger number of ants appears to produce more visually appealing results, with 1000-2000 ants being the optimal range; however, the size of the terrain may have an influence over this number.

2. Higher `slopeWeight` values seem to, in fact, help ants navigate the landscape.

3. Extremely low values for evaporation and diffusion lead to better defined ant trails which in turn helps wandering ants find food.

Lastly, Experiment 615's parameter values were copied over to a new experiment and tested on a real-life based terrain, which served as a preliminary experiment on custom landscapes (rather than being flat or generated via Perlin noise), with images on the previous chapter showcasing the terrain after the same 10 000 iterations from two different view angles. This time, ants seem to have accumulated around the central area of the landscape rather than spread out, but seem to have the same tendencies of aiding others find their path towards food sources (trails seem to be less visible due to the amalgamation of ants, but the presence of red *paint* indicates otherwise). The main conclusions from this last experiment were that the introduction of custom landscapes is a viable option in the simulation of ants with this specific algorithm (which could prove extremely interesting if these landscapes were to be copies of slices of a real-life ants' habitat) and tweaks to the excavation values (`foodHeightIncr` and `noFoodHeightIncr`) are necessary to get good visual results and prevent some visualization issues found in the current version of this algorithm (this, however, applies to all experiments, not just the ones being made with custom terrains).

5.2 Simulated Landscape Dynamics

In nature, ecosystems, whether vast or small, transform over time as species evolve and adapt to their surroundings, reshaping the landscape in the process. It was this idea of continual transformation that inspired the algorithm's design: to create agents that emulate the behaviours of a specific species, and through these behaviours, modify the environment they inhabit.

Inspired by biological studies on ants, the algorithm simplifies ant behavior to its essentials. Only the key actions (digging, food gathering, and basic wandering) were implemented, adhering to straightforward, locally executed rules. This minimalistic approach maintains fidelity to a somewhat real-world colony behaviour while allowing the model to operate effectively through parameterizable-driven actions.

The algorithm is also guided by the Principle of Mass Conservation, which states that matter is neither created nor destroyed, only relocated. This principle ensures that any terrain removed by a moving ant is distributed to neighbouring cells. By using fixed redistribution values (see Figure 3.1), the model transforms the landscape consistently without 'losing material', supporting stable, ongoing landscape modification rather than destruction.

Departing from typical Ant Colony Optimization (ACO) models, which often rely on graph-based approaches to simulate movement and decisions, this algorithm operates within a continuous 2D grid. While most simulations restrict themselves to strictly two-dimensional dynamics, this model introduces *height-awareness*, giving it added depth.

The ants, navigating a 2D grid and equipped with this form of *height perception* are able to both detect and alter elevation, effectively bridging the gap between a 2D simulation and a simplified 3D space. While moving in the horizontal X and Z axes, their interactions affect elevation on the vertical Y axis. This dynamic interaction of 2D movement with 3D terrain changes enables complex landscape transformations, balancing detail with computational demands.

All simulations were based on a grid of 263 169 cells, representing all available spaces in the landscape, with food occupying just about 0.000 38% of this space (in experiments with up to 100 sources). This sparse food distribution, reflecting the resource scarcity often found in natural environments, encourages the ants to disperse, simulating the challenges of foraging across large areas. This setup also enables observation of how simple, localized agent interactions can scale up to produce impactful, landscape-wide changes.

In Experiment 4, the effects of varying the amounts of food and ants were examined, with some interesting contrasts to natural systems. For example, the number of ants and food was increased well beyond what would likely be sustainable in a real-world environment, revealing how the model might respond under hypothetical or extreme conditions. This exploration was useful to understand the algorithm's flexibility and the resulting visual landscape modifications, even if the setup diverged somewhat from natural conditions.

5.3 Limitations

Although the algorithm did show interesting results, it did not come to be without some drawbacks regarding its implementation. One of such drawbacks was the fact that the redistribution of terrain height contained within the heightmap was based on fixed percentages. As seen in Figure 3.1, when an ant moves, the removed height value of an ant's current position is distributed to some of its surrounding positions based on its movement pattern. However, although the total sum of height values in percentage does, in fact, add up to 100%, the percentage that each location receives never changes (as long as the

movement pattern also never does). It's not hard to guess that changing these values would lead to the generation of vastly different landscapes and thus parameterising them could be a great plan for the future. A better idea, however, would be the implementation of a *safeguard* algorithm that would distribute height values based on their own individual surroundings, in order to combat height values becoming ridiculously high and ruining a landscapes' presentation (as seen in Figure 5.1).

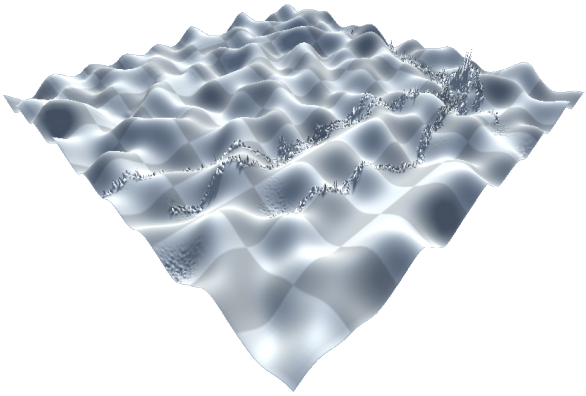


Figure 5.1: Example of an experiment whose values lead to a very visually unappealing landscape. Notice the *spikes* along the ants' trails and their accumulation around the colony (on the right).

Another drawback (somewhat related to the one previously discussed) was the reliance on using only the nearest neighbouring cells around an ant to redistribute height levels. The implementation of a bigger affected area could be an interesting (although not as realistic) idea to play with, since a single ant's path would be able to *cover* a lot more space than currently.

One notable issue that had to be fought was the definition of a proper ceiling value for the terrain's height. Unity's `Terrain` objects contain this information in their `Terrain Height` variable, which represents a landscape's available space in the Y axis (in world units). This value can be programmatically manipulated in real-time although doing so will stretch/squash the heightmap, leading to manipulated final results and an incorrect display of the values the algorithm is working with. To get around this issue, a small static value (5) was selected for the variable, which, together with the heightmap's set resolution and ant digging values proportioned correct results for most run experiments. A possible solution could be as simple as increasing this value to accommodate more space before the algorithm is run (taking into account the maximum value allowed by Unity is 10 000), although several tests would need to be performed in order to confirm that Unity would not be modifying heightmap values internally (i.e., the values the ants would be working with would need to correspond 1:1 with the ones being displayed).

There were also behavioural drawbacks regarding the ants themselves, mainly the pilling up of such agents within a small radius surrounding their colony. As ants modify the scenery, they inadvertently end up influencing others' path making choices, since their movement is dictated by their surroundings. This, combined with the fact that all ants share the same colony leads to the accumulation of such agents around this particular hot spot (reminding that the colony's function is that of a food deposit). A way to tackle this problem could be the assignment of several groups of ants to multiple colonies, while making

sure their locations are well spread out (this may not necessarily fix the ants' behaviour, but it would certainly help in distributing them more across the landscape, thus lowering the chances of trapping them around a singular location).

Chapter 6

Conclusions

This chapter contains a brief overview of the achieved goals and the work ahead.

6.1 Achievements

This dissertation explored an innovative method for creating virtual landscapes through the use of a PCG algorithm inspired by ant-like agent behaviour. It sets the ground work for a new type of offline terrain generation procedure, whose experimental scenarios helped demonstrate its capacity in creating varied and interesting terrains.

Extensive testing demonstrated the ants' trail formation capabilities across a range of values assigned to various behaviour variables, as well as their ability to navigate vast landscapes, whether based on Perlin noise, simple flat surfaces or real-life based scans. Pheromone placement, diffusion and evaporation also showed positive results regarding ants' actions, correctly influencing their tracking abilities and allowing them to find points of interest previously accessed by others.

Overall, this project succeeded in its purpose, offering insights into a scalable, biologically inspired solution for virtual environments. Its use is aimed toward video games, although its applications could extend beyond entertainment to fields like simulation training, environmental modelling, and virtual reality experiences. By simulating natural trail formation and navigation, the project demonstrates a method that could enhance realism and variability in digital landscapes, making them more engaging and adaptable to user interactions.

6.2 Future Work

Many of the possible improvements to this project are tied to the algorithm's current limitations. Thus, the amount of *dirt* ants are able to move, the terrain visualization issues within Unity, and the accumulation of ants near points of interest are all interesting topics to tackle in order to improve behavioural patterns and visual limitations.

Other appealing ideas that should be explored relate to characterization of the ants themselves. In

this dissertation, ants' behaviours are based upon working ants, whose purpose is the collection of food, benefiting the colony. The incorporation of other types of ants could pose a yet underachieved challenge as *ant warfare* is a complex topic by itself.

Finally, optimizing the current algorithm also poses an exciting challenge. The creation of a faster, real-time solution supported by parallelization methods could certainly enhance its ability to handle larger ant colonies as well as terrain sizes. These improvements would not only elevate performance but also strenghten the project's overall scalability and future potential.

Bibliography

[1] Saman Almufti, Awaz Shaban, Rasan Ali, and Jayson Fuente. 2023. Overview of Metaheuristic Algorithms. *Polaris Global Journal of Scholarly Research and Trends* 2 (04 2023), 10–32. https://doi.org/10.58429/pgjsrt.v2n2a144

[2] Nicolas A. Barriga. 2019. A Short Introduction to Procedural Content Generation Algorithms for Videogames. *International Journal on Artificial Intelligence Tools* 28, 2 (2019). https://doi.org/10.1142/S0218213019300011

[3] Farès Belhadj. 2007. Terrain Modeling: A Constrained Fractal Model. In *Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. 197–204. https://doi.org/10.1145/1294685.1294717

[4] B. Beneš and X. Arriaga. 2005. Table mountains by Virtual Erosion. In *Proceedings of the First Eurographics Conference on Natural Phenomena* (Dublin, Ireland) *(NPH'05)*. Eurographics Association, 33–40. https://doi.org/10.5555/2381356.2381362

[5] Diogo de Andrade, Nuno Fachada, Carlos M. Fernandes, and Agostinho C. Rosa. 2020. Generative Art with Swarm Landscapes. *Entropy* 22, 11 (Nov. 2020), 1284–1284. https://doi.org/10.3390/e22111284

[6] Tansel Dokeroglu, Ender Sevinc, Tayfun Kucukyilmaz, and Ahmet Cosar. 2019. A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering* 137 (2019). https://doi.org/10.1016/j.cie.2019.106040

[7] Marco Dorigo. 1992. *Optimization, learning and natural algorithms*. Ph.D. Thesis. Politecnico di Milano.

[8] David S Ebert. 1996. Advanced modeling techniques for computer graphics. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 153–156.

[9] Wim Eck and Maarten Lamers. 2015. Biological Content Generation: Evolving Game Terrains Through Living Organisms, Vol. 9027. https://doi.org/10.1007/978-3-319-16498-4_20

[10] Kati Steven Emmanuel, Christian Mathuram, Akshay Rai Priyadarshi, Rishu Abraham George, and J Anitha. 2019. A Beginners Guide to Procedural Terrain Modelling Techniques. In *2019 2nd*

International Conference on Signal Processing and Communication (ICSPC). 212–217. https://doi.org/10.1109/ICSPC46172.2019.8976682

[11] Nuno Fachada, António R. Rodrigues, Diogo de Andrade, and Phil Lopes. 2024. Generating 3D Terrain with 2D Cellular Automata. https://doi.org/10.48550/arXiv.2406.00443

[12] Carlos M Fernandes. 2010. Pherographia: Drawing by ants. *Leonardo* 43, 2 (2010), 107–112.

[13] Carlos M Fernandes, Carlos Isidoro, Fábio Barata, Agostinho C Rosa, and Juan Julián Merelo. 2011. From pherographia to color pherographia: Color sketching with artificial ants. In *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, 1124–1131.

[14] Alain Fournier, Don Fussell, and Loren Carpenter. 1982. Computer rendering of stochastic models. *Commun. ACM* 25, 6 (June 1982), 371–384. https://doi.org/10.1145/358523.358553

[15] Hello Games. 2016. No Man's Sky.

[16] Marko Gulić, Martina Žuškin, and Vilim Kvaternik. 2023. An Overview and Comparison of Selected State-of-the-Art Algorithms Inspired by Nature. *TEM Journal* 12 (08 2023), 1281–1293. https://doi.org/10.18421/TEM123-07

[17] Fatih Gürler and Esin Onbaşioğlu. 2022. Applying Perlin Noise on 3D Hexagonal Tiled Maps. In *2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. 670–673. https://doi.org/10.1109/ISMSIT56059.2022.9932712

[18] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications and Applications* 9, 1 (Feb. 2013), 22 pages. https://doi.org/10.1145/2422956.2422957

[19] Shuo Huang and Xiang-Xin Li. 2010. Improved Random Midpoint-Displacement Method for Natural Terrain Simulation. *2011 Fourth International Conference on Information and Computing* 1 (2010), 255–258. https://doi.org/10.1109/ICIC.2010.71

[20] Dervis Karaboga. 2005. *An idea based on honey bee swarm for numerical optimization.* Technical Report. Technical Report tr06, Erciyes University.

[21] George Kelly and Hugh McCabe. 2006. A Survey of Procedural Techniques for City Generation. *The ITB Journal* 7, 2 (2006), 5.

[22] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95–International Conference on Neural Networks*, Vol. 4. 1942–1948.

[23] Eng-Kiat Koh and D. D. Hearn. 1992. Fast Generation and Surface Structuring Methods for Terrain and Other Natural Phenomena. *Computer Graphics Forum* 11, 3 (1992), 169–180. https://doi.org/10.1111/1467-8659.1130169

[24] David Maung, Yinxuan Shi, and Roger Crawfis. 2012. Procedural textures using tilings with Perlin Noise. In *2012 17th International Conference on Computer Games (CGAMES)*. 60–65. https://doi.org/10.1109/CGames.2012.6314553

[25] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. 2007. Fast Hydraulic Erosion Simulation and Visualization on GPU, In 15th Pacific Conference on Computer Graphics and Applications. *Pacific Graphics*, 47–56. https://doi.org/10.1109/PG.2007.15

[26] Panagiotis A. Mikedakis. 2017. *Two and Three-Dimensional Cellular Automata for the Generation of Objects in Computer Graphics*. Bachelor's Thesis. National and Kapodistrian University of Athens.

[27] Jacob Olsen. 2004. Realtime Procedural Terrain Generation. https://api.semanticscholar.org/CorpusID:18949321

[28] Ken Perlin. 1985. An Image Synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, New York, NY, USA, 287–296.

[29] Pascale Roudier, Bernard Péroche, and M. Perrin. 1993. Landscapes Synthesis Achieved through Erosion and Deposition Process Simulation. *Computer Graphics Forum* 12 (1993). https://api.semanticscholar.org/CorpusID:29027868

[30] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. 2009. A Survey of Procedural Methods for Terrain Modelling. In *Proceedings of the CASA 2009 Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*. 25–34.

[31] Gillian Smith, Elaine Gan, Alexei Othenin-Girard, and Jim Whitehead. 2011. PCG-based game design: enabling new play experiences through procedural content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games (PCGames '11)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/2000919.2000926

[32] Tommaso Toffoli and Norman Margolus. 1987. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/1763.001.0001

[33] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. 2011. What is procedural content generation? Mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games (PCGames '11)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2000919.2000922

[34] Julian Togelius, Georgios Yannakakis, Kenneth Stanley, and Cameron Browne. 2010. Search-Based Procedural Content Generation. 141–150. https://doi.org/10.1007/978-3-642-12239-2_15

[35] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane. 1980. Rogue.

[36] Unity Technologies. 2005. Unity Game Engine. Software available at https://unity.com. Version used: 2021.3.6f1.

[37] Fevrier Valdez. 2021. *Swarm Intelligence: A Review of Optimization Algorithms Based on Animal Behavior*. 273–298. https://doi.org/10.1007/978-3-030-58728-4_16

[38] Thomas A Witten Jr and Leonard M Sander. 1981. Diffusion-limited aggregation, a kinetic critical phenomenon. *Physical review letters* 47, 19 (1981), 1400.

[39] Don Worth. 1978. Beneath Apple Manor.

[40] Georgios N. Yannakakis and Julian Togelius. 2018. *Artificial Intelligence and Games*. Springer. https://gameaibook.org/

[41] Yuzhong Zhang, Guixuan Zhang, and Xinyuan Huang. 2022. A Survey of Procedural Content Generation for Games. In *2022 International Conference on Culture-Oriented Science and Technology (CoST)*. 186–190. https://doi.org/10.1109/CoST57098.2022.00046

[42] Peter Ziegler and Sebastian von Mammen. 2020. Generating Real-Time Strategy Heightmaps using Cellular Automata. 1–4. https://doi.org/10.1145/3402942.3402956