UNIVERSITY OF CANBERRA

Generating a Virtual Forest Environment using Procedural Content Generation

Liam Potter

u3140143

A thesis submitted in fulfilment for the Bachelor of Arts and Design (Honours)

Faculty of Creative Arts and Digital Design

Canberra University

November 2017

Acknowledgements

I'd like to give special thanks to Dr. Reza Ryan, Matt McRae, Wade Taylor, Chris Osmond and my family for their help and support throughout the development of this research project.

Keywords

Game Content Generation, Forest Generation, Real-Time, Procedural, Virtual Environment, Forest Visualisation, Simulation.

Abstract

The aim of this research was to create a system that would generate a virtual forest in real-time. Over the past three decades the amount of content needed to produce virtual forests have become immense because of the changes to video game worlds. In this research, a system was developed to integrate different Procedural Content Generation techniques to automatically generate and simulate a virtual forest in real-time. These techniques include Height Generation, Terrain Texture Generation, Detail Generation, Point Generation, Shadow Map Generation, Life Cycle Simulation and Day/Night Simulation. These techniques were then integrated and implemented into a Video Game Engine. The developed system can be easily integrated with any real-time game that requires a forest environment and the techniques developed are easily modifiable.

Table of Contents

List of Figures

List of Abbreviations

Procedural Content Generation (PCG)

Video Game Engine (VGE)

Three Dimension (3D)

Two Dimension (2D)

Lindenmayer system (L-System)

Frames Per Second (FPS)

Graphics Processing Unit (GPU)

Garbage Collection (GC)

Life Cycle (LC)

Design Science Research (DSR)

Design Science Research Method (DRSM)

Chapter 1: INTRODUCTION

As the target audience of video games gets larger, so too does the worlds within them. Video game worlds have evolved from small, individual levels to massive sprawling environments over the span of the past three decades. Because of these changes to video game worlds, the sheer amount of content digital artists need to produce is immense. To combat this avalanche of content, game development companies have developed automated techniques for generating content. These are known as procedural content generation(PCG) techniques, and they can be applied to generate a wide variety of things, from entire cities to each individual leaf on a tree (Hendrikx, Meijer, Van Der Velden & Iosup, 2013).

1.1 Research Problem

Virtual forests are one of the more common and complex virtual environments in contemporary video games that need to be generated procedurally. These complex virtual environments require many objects to populate them, ranging from small mushrooms to century old trees. The geographical aspects of real forests can differ greatly, so any virtual representation of them needs to be very flexible in their implementation. Each tree in a forest should be similar in many aspects to its neighbours, yet also requires some sense of individuality (Ch'ng, 2011).

Virtual forests need to be broad and dense while maintaining a high level of detail and aesthetic realism (Ch'ng, 2011). The amount of time taken to create forest environments by hand is costly for bigger development companies, and simply un-maintainable for smaller development companies (Hendrikx et al, 2013). To combat this, procedural content generation techniques can be employed to generate the many parts of a forest environment. However, contemporary procedural content generation techniques need to be altered and controlled by certain rules. In addition, they generally only place static objects in virtual space, with nothing affecting the objects after their initial placement. The life-cycle simulation of an entire forest environment is something needed for visualisation and increased user immersion (Li, Zhang, Meng & Ge, 2017).

1.2 Problem Statement

Through research and analysis of contemporary procedural content generation techniques, this project plans to prototype a system capable of generating and simulating large, virtual forest environments in real-time. This project will adhere to an objectivist standpoint in conjunction with a theoretical vision comprising of reliability, sustainability, complexity and customizability when collecting data, analysing data and creating prototypes.

1.3 Contribution to the Body of Knowledge

While many contemporary content generation algorithms are impressive on a large scale, they lack aesthetic realism at closer observation (Hendrikx et al, 2013; Kelly, 2008; Müller et al, 2006). In addition to this, they are rarely designed for use in real-time applications. As such, this project's contribution to the body of knowledge is the development and documentation of content generation and simulation algorithms that can generate and simulate virtual forests with a high degree of detail and realism in real-time. The realism factor of a virtual environment is important as it increases user immersion into the product.

To achieve a virtual forest with all these required objects and characteristics, it is proposed that different procedural content generation techniques be combined. Certain optimisations have been taken into account when combining the techniques to get the autonomous forest rendered at real-time. Real-time means the forest environment will be rendered and updated approximately in sixty frames a second. The autonomous forest will be developed in the Video Game Engine (VGE) Unity. The product will procedurally generate entire forests autonomously, from the overall shape of the environment, to the simulation of floral growth.

1.4 Research Question

This research will be generating a virtual forest environment using procedural content generation. Identification of the research gaps has led to the formulation of the following research questions.

- What are the required methods and features for generating a virtual forest environment using procedural content generation (PCG)?

- How can different PCG systems be incorporated into a unique method of fully generating and simulating a virtual forest environment in real-time?

- How can the proposed system be analysed and evaluated using the Unity engine?

1.5 Research Aims and Objectives

This project aims to investigate and develop various procedural content generation and real-time simulation algorithms and apply them appropriately in a working prototype. This prototype will consist of a system that can generate forests, including, but not limited to, trees, rocks, fungi and shrubbery. The prototype will then simulate the life-cycle of those objects before realistically simulating a day/night cycle using sun angles.

Two objectives have been produced for the outcome of this research project:

1. Present a fully designed concept that effectively addresses the research problems
2. Produce a working prototype that showcases all the research and design taken place within the project

1.6 Scope and limitations

For this project to meet its goals in a timely manner, both the scope and limitations of the project must be clearly understood and adhered to.

This project will be limited to the following:

- Landscape simulation and evaluation,
- Flora and Fungi placement, scale and life-cycle simulation
- Day/Night simulation

This project will not be doing the following:

- It will NOT be generating the 3D models used in the generation algorithm
- It will NOT be populating the environment with wildlife

1.7 Publication

The findings of this research were accepted as a full conference paper:

Potter, L. (2017). Generating a Virtual Forest Environment using Procedural Content Generation. In *CreateWorld*. Brisbane: Apple University Consortium.

1.8 Outline of the Report

This report comprises five chapters. Following this introductory chapter is the literature review, Chapter 2, which provides critical analysis of related work in the field of Procedural Content Generation.

Chapter 3, Generating a Virtual Forest Environment, provides details for the research methodology and technical concepts of the proposed approach.

Chapter 4, Prototyping, Results, and Evaluation, gives details of the project's implementation, experiments result and evaluation,

Finally, Chapter 5, the conclusion, summarises the research findings and proposes future works.

Chapter 2: RELATED WORKS

This chapter critically evaluates the existing knowledge in the research area to identify research gaps. This chapter is divided into four sections; Distribution and Placement, Poisson Disk Sampling, Plant Ecosystem Modelling and Rendering and the gap in the literature. The first three sections present areas of Procedural Content Generation and realistic floral ecosystems. The last section covers the research gap identified in the previous, as well as the planned approach to fill that gap.

2.1 Distribution and Placement

There are numerous ways of implementing procedural distribution and placement, including Poisson Disc sampling (Kailkhura, Thiagarajan, Bremer & Varshney, 2016; Park, Pan & Manocha, 2014; Park, Pan & Manocha, 2017; Ying, Xin, Sun & He, 2013) and Realistic Spatial Distribution (Lane & Prusinkiewicz, 2002). Kailkhura et al. (2016) found that Poisson Disc sampling is an exceptional method for quick, uniformed and random point placement. This view is supported Ebeida, Mitchell, Patney, Davidson and Owens (2012, p. 2), who state "This pattern yields good visual resolution for rendering, imaging, texture generation, and geometry processing."

Poisson Disc sampling is a procedural point placement algorithm that evaluates each placed point and makes sure they are a certain distance away from their neighbours (Ying et al, 2013). This leads to a random, yet uniform distribution that provides a useful data set for many different applications such as "anti-aliasing, global illumination, non-photorealistic rendering, re-meshing, texture synthesis, and vector field visualization (Ying et al, 2013). In comparison with the other procedural placement techniques, Poisson Disc sampling is the optimal algorithm when requiring evenly yet randomly distributed points as techniques like Uniform Random Distribution and Jittered Grids possess too many opportunities for unwanted clumping (Figure 1).

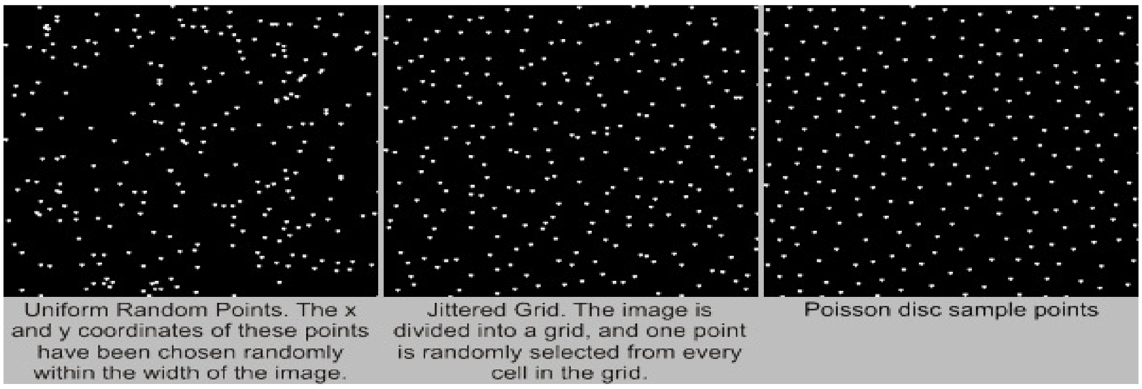
Uniform Random Points. The x and y coordinates of these points have been chosen randomly within the width of the image.
Jittered Grid. The image is divided into a grid, and one point is randomly selected from every cell in the grid.
Poisson disc sample points

Figure 1 Poisson Disc sampling figure (Tulleken, H. 2009)

Another solution to procedural object placement provided in a paper by Lane and Prusinkiewicz (2002). This paper dives into extremely realistic plant placement via multiset L-systems, with each plant spreading their own seeds into the virtual environment. An L-System, otherwise known as a Lindenmayer system is type of formal grammar, consisting of an alphabet of user-defined symbols. The L-System adheres to specific 'production rules' that are used to expand each individual symbol into larger sets of strings. Then, after generating these strings the L-System employs a mechanism to translate them into geometric shapes and structures (Suhartono, Hariadi & Purnomo, 2012).

Lane and Prusinkiewicz (2002) cleanly defined the two approaches that they would cover in their own research. The first approach was defined as the local-to-global approach, meaning the placement and distribution of plant densities was controlled by simulations of interactions between individual plants. The second approach, global-to-local, was distinguished as controlling plant positions by inferring from large-scale density distributions.

Based on Lane and Prusinkiewicz (2002) analysis of procedural plant placement, they suggested a system that placed plants randomly in a virtual environment and iteratively 'grew' them, and 'killed' them when dominated by larger plants. Lane and Prusinkiewicz (2002, p. 2) claimed that "the resulting distribution fit the self-thinning curve of plant ecology, a relationship between the average mass and average density in a monoculture of plants of the same age." This 'growth & death' system was developed further to implement the previously mentioned local-to-global approach. To implement the other proposed approach, global-to-local, Lane and Prusinkiewicz (2002) employed the Floyd-Steinberg error diffusion algorithm alongside a singular grayscale image to create each individual plant position.

The Floyd-Steinberg error diffusion algorithm is an image dithering algorithm, meaning it intentionally employs a type of noise to a supplied image, preventing large scale colourbanding and other unwanted patterns that may occur within the image. This algorithm is commonly used for converting grayscale images into black and white images, and converting images into image formats that are restricted to 256 colours (Figure 2).

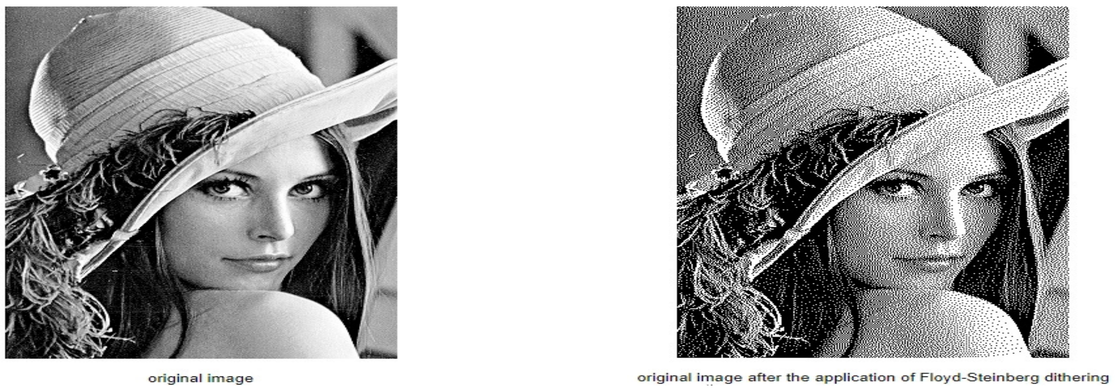
original image
original image after the application of Floyd-Steinberg dithering

Figure 2 Images before and after applying Floyd-Steinberg dithering (VISGRAF Lab, 2000)

While both local-to-global and global-to-local systems were fine in terms of placing arbitrary points in a random yet controlled manner, they both lacked something apparent in real-world ecosystems, clustering. Lane and Prusinkiewicz, (2002, p. 2) define clustering, also known as clumping or under dispersion, as “a common phenomenon, caused by environmental factors (plants of the same type tend to cluster in the areas favourable to their growth), propagation (seeds fall close to their parent plants, or plants propagate by runners), as well as other mechanisms.”

They state that clustering has a significant impact on the appearance of any plant distribution, causing their interest into modelling it. To simulate this phenomenon, Lane and Prusinkiewicz (2002) utilized the Hopkins Index, which is an algorithm that calculates the ratio of the distance from a tree to its nearest neighbour, as well as the distance from random points within the same space as that neighbouring tree as shown in the equation below. (Lane & Prusinkiewicz, 2002; Natural Resource Biometrics, 2014)

$$H = \frac{\langle min_i(\|x - p_i\|)\rangle x}{\langle min_i(\|p_j - p_i\|)\rangle j}$$

In the above equation, the value H is cluster tendency. A value of H close to 1 tends to indicate the data is highly clustered, and uniformly distributed data will tend to result in values of H close to 0. The value x is the desired distance between points. Value p_i is the initial point being used in the distance calculation and p_j is a random point near p_i.

One of the major limitations of the system Lane and Prusinkiewicz (2002) developed was the time it took to generate and render a single scene. Their generation and render times ranged from 8 minutes to 66 minutes when rendering a single image of their scene. This makes it apparent that Lane and Prusinkiewicz's (2002) system was not designed and developed with real-time optimization taken into account.

2.2 Poisson Disk Sampling

Bridson (2007) produced a paper on the creation and implementation of a randomized point placement system that can place points in 3D space. To be more specific, Bridson covered the Poisson Disk Sampling algorithm to achieve a blue noise effect, with a random, yet structured point distribution. Bridson (2007, p. 1) explains that "In many applications in graphics, particularly rendering, generating samples from a blue noise distribution is important". Similarly, Yan, Guo, Wang, Zhang and Wonka (2015, p. 1) state "In computer graphics, sampling plays an important role in many applications, such as rendering, stippling, texture synthesis, object distribution, and simulation."

Bridson's (2007) research states that though blue noise generation techniques may exist, they do not easily translate from two dimensions to three. This is because almost all previous blue noise algorithms were designed to be applicable to other image sampling techniques Bridson (2007). Bridson's (2007) solution is based on a previous algorithm that placed points on a plane between r(the minimum distance between points) and $2r$. They iterate on this algorithm to implement it into three dimensions by doing the following steps:

Step 0: Initialize an n-dimensional background grid for storing samples and accelerating spatial searches. Then, pick the cell size to be bounded by $r/\sqrt{n}$, so that each grid cell will contain at most one sample, and thus the grid can be implemented as a simple n-dimensional array of integers: the default -1 indicates no sample, a non-negative integer gives the index of the sample located in a cell.

Step 1: Select the initial sample, x_0, randomly chosen uniformly from the domain. Insert it into the background grid, and initialize the "active list" (an array of sample indices) with this index (zero).

Step 2: While the active list is not empty, choose a random index from it (say i). Generate up to k points chosen uniformly from the spherical annulus between radius r and $2r$ around x_i. For each point in turn, check if it is within distance r of existing samples (using the background grid to only test nearby samples). If a point is adequately far from existing samples, emit it as the next sample and add it to the active list. If after k attempts no such point is found, instead remove i from the active list.

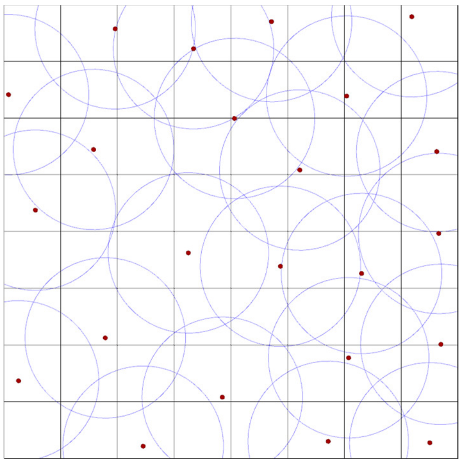

Figure 3 Poisson Disk Sampling Implementation

Implementation of the algorithm presented by Bridson (2007) can be seen in Figure 3. As seen in the figure, all points generated by the algorithm are placed a certain distance from all other points, illustrated by the blue circles around each point. The algorithm will place only one point in each grid section, but will ignore any section that is completely overlapped by blue circles, meaning there is no possible location in these sections a point could be placed.

The main weakness with this Poisson Disk Sampling method is the possibility for slight gaps to occur within the final placement. This is due to the first point in the algorithm being chosen randomly within the entire grid. The gaps have been highlighted in red in Figure 4 and as seen the gaps only occur on the left side of the grid, which is where the initial point was randomly placed. These errors do not affect the overall placement of the rest of the points, but they are worth noting.

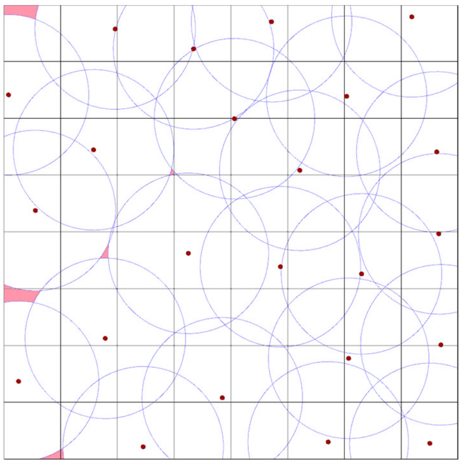

Figure 4 Poisson Disk Sampling Implementation Errors

2.3 Plant Ecosystem Modelling and Rendering

Deussen, Hanrahan, Lintermann, Měch, Pharr and Prusinkiewicz (1998) produced a study into the best method for modelling and rendering a realistic plant ecosystem. Duessen et al. have listed the five main techniques they used to design and implement the plant ecosystem, these techniques are; A multilevel modelling and rendering pipeline, open system architecture, procedural models, approximate instancing and efficient rendering.

A multilevel modelling and rendering pipeline was used to divide the system Duessen et al. produced. This was useful as it allowed the user to focus on one segment of the system without worrying about how it will affect the others. Deussen, Hanrahan, Lintermann, Měch, Pharr and Prusinkiewicz (1998, p. 3) notes that "Thus, the modeler is not concerned with plant distribution when specifying the terrain, and plant distribution is determined (interactively or algorithmically) without considering details of the individual plants"

An open system architecture was implemented in the project, allowing for the incorporation of independent modules. These modules would be able to affect any part of the system, thus allowing for easy implementation of new plant models, different placement methods and generation approaches.

Procedural models are usually defined by substantial data-base amplification, meaning they can create and generate complicated geometric structures from a small amount of inputted data. Duessen et al. (1998, p. 3) stated "We benefit from this phenomenon by employing procedural models in all stages of the modelling pipeline."

Approximate instancing, a combination of object instancing and approximated clustering, was incorporated into the project as the main method for reducing the end file size of any rendered scenes. To optimize the amount of instancing, objects and their components were clustered in their parameter space.

After this approximated clustering, objects within any given cluster was rendered with instances of a singular representative object. Duessen et al. (1998, p.3) extended this system by "applying vector quantization to find the representative objects in multidimensional parameter spaces."

Finally, Duessen et al. (1998) used memory and time efficient rendering techniques to speed up to time it took to render their generated scenes. These techniques included: the dissection of the generated scene into sub-scenes for composition in later steps, ray-tracing with instancing support for rendering many polygons and memory-coherent ray tracing applicable to instanced objects.

By applying these techniques to their project, Duessen et al. (1998) generated and rendered a scene that held up to 100,000 highly detailed objects. This number could be further increased, however Duessen et al. (1998, p. 3) stated that "with 100,000 plants, each plant is visible on average only in 10 pixels of a 1K pixel by 1K pixel image. Consequently, we seem to have reached the limits of useful scene complexity, because the level of visible detail is curbed by the size and resolution of the output device." While Duessen et al. (1998) created an extensive system for generating plant ecosystems; their output was severely limited to the rendering of a single image, thus removing any potential interaction and immersion from their project.

2.4 The Gap in the Literature

This literature review identifies a situation that while many contemporary content generation algorithms and techniques can create large, immersive and intricate environments they are rarely designed and optimised to run in real-time. Therefore, this research will create and implement content-generation algorithms that can generate and simulate immersive virtual forests in real-time.

To achieve this goal, this research will incorporate seven different systems: Height Generation, Terrain Texture Generation, Detail Generation, Point Generation, Shadow Map Generation, Life Cycle Simulation and Day/Night Simulation. Through the combination of these systems, this research will provide a unique method of fully generating and simulating a virtual forest environment in real-time.

Chapter 3: GENERATING A VIRTUAL FOREST ENVIRONMENT

This chapter discusses this project's research methodology and optimization tactics before discussing the seven generation and simulation systems that will be implemented in this project. This chapter is divided into nine sections, with the first section addressing the project's research methodology and the second section explaining the real-time optimization techniques implemented in this project. Each section past the second address the theoretical aspects and algorithms behind one of the seven systems developed within the project.

3.1 Methodology

Due to the intrinsic nature of this research it has been concluded that using Design Science Research (DSR) methodology to develop each phase is the optimal methodology for this project.

DSR is used to develop general knowledge which can be used to design solutions to the specific problems (Chatterjee, Peffers, Rothenberger, Tuunanen, 2007). One of the common methodological approaches used by this method is Design Science Research Method (DRSM). The DRSM process includes 6 activities, all of which will be incorporated into this project.

As shown in Figure 5, this project started with problem identification and motivation, which defined the specific research and justified the value of the solution. By a performing preliminary survey on the current state Procedural Content Generation research, it was possible identify what research has already been undertaken around the research problem, as well has define the significance of this project's research.

Next, it was defined what exactly the project's solution was trying to accomplish. To achieve this, a literature review was performed on the current field of Procedural Content Generation. The solution's objective was the creation, modification and implementation of generation and simulation algorithms to generate and simulate a virtual forest in real-time. This objective showed how the project was expected to support its research problem and addresses the research question the project is presenting.

The third step, as shown in Figure 5, was design and development, in which the project developed generation and simulation algorithms as well as modified existing PCG algorithms to suit the artefact. This step encompassed all the theoretical development that was used later in practical implementations. This activity determined the artefact's desired functionality and architecture before actual implementation.

The next step was demonstration, in which the project showcased the created artefact and demonstrated its feasibility and ability to address the research question. The demonstration in this project was the full implementation of the designed algorithms. This was the main practical aspect of the project, and was the basis for any further iteration and analysis. By documenting the implementation process, it was possible to determine how affective the algorithms were at their various tasks.

The fifth step in Figure 5 was the evaluation of the created artefact. This activity was used to observe and measure how well the artefact supported a solution to the research problem. This was the main analytical step of the research that reviewed both the design and implementation of the created algorithms. The artefact was evaluated using a detailed scenario to demonstrate its functionality and utility. This step was the project's main point of iteration, in which the artefact was analysed for its feasibility, function and utility. After deciding the artefact needed iteration, the project stepped back into the third step, design and development, to rework the designed algorithms and systems.

The last step taken was the communication of the research problem and its significance in the field it was stationed within. In addition, the artefact was also communicated about with its utility, design and effectiveness. This step was incorporated into the whole paper as it followed proper scholarly practice to effectively communicate the research. This activity was performed with this dissertation and a published paper variant of the research.

With these six steps, it was possible to plan, construct and evaluate the research with the advancement of scientific research in mind. By doing this, the research holds intrinsic value in the field of both Procedural Content Generation and various real-time applications. The next section will be an evaluation and explanation of the various optimization techniques incorporated into the theoretical and practical aspects of this project.

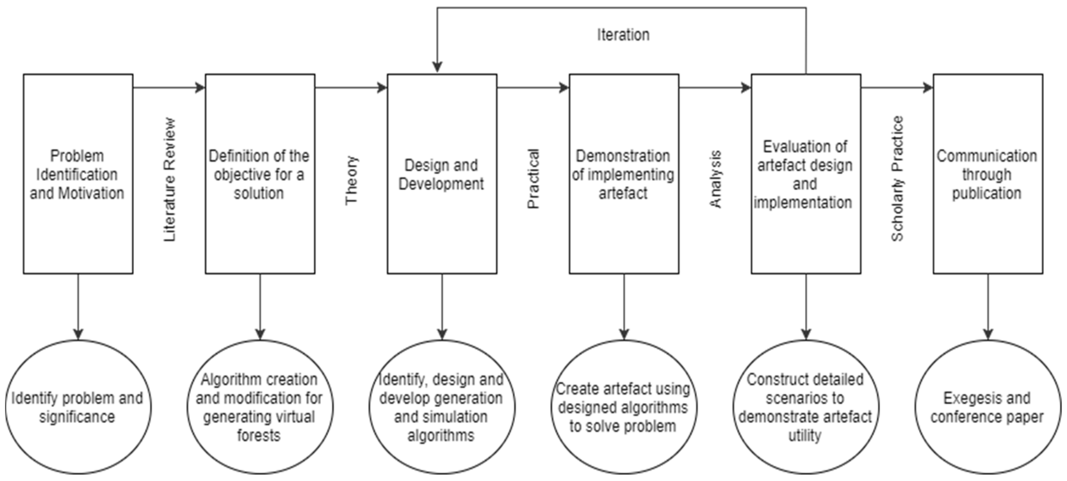
Iteration
Problem Identification and Motivation
Literature Review
Definition of the objective for a solution
Theory
Design and Development
Practical
Demonstration of implementing artefact
Analysis
Evaluation of artefact design and implementation
Scholarly Practice
Communication through publication
Identify problem and significance
Algorithm creation and modification for generating virtual forests
Identify, design and develop generation and simulation algorithms
Create artefact using designed algorithms to solve problem
Construct detailed scenarios to demonstrate artefact utility
Exegesis and conference paper

Figure 5 The Design Science Research Methodology

3.2 Mass Object Rendering

One major constraint in simulating an entire virtual forest environment lies in the substantial amounts of objects that need to be rendered on screen at once. This issue is further amplified by the fact that the virtual forest this project will be generating needs to be rendered 60 times a second. Each of these renders are referred to as 'frames', and the rate at which these frames are updated is called 'Frames Per Second' (FPS). This project intends to render the generated virtual forest environment at a minimum of 60 FPS.

In contemporary Video Game Engines (VGEs), each rendered frame is broken down into 'draw calls'. These draw calls refer to any object that needs to be 'drawn' on each frame; the more objects drawn, the slower it takes to render a frame. In addition to draw calls, the FPS of a project can also be negatively affected by un-optimized scripts. To increase the FPS of a project, there are a few techniques that can be employed within VGE. These include, but are not limited to, Graphics Processing Unit(GPU) Instancing, Occlusion Culling and Garbage Collection(GC) optimization.

The first technique this project will be implementing is GPU Instancing, which renders multiple copies of a mesh simultaneously, thus reducing the amount of draw calls needed. The main usage of GPU instancing is when multiple copies of the same mesh is placed throughout the scene, making it a viable option for a forest environment as they tend to be filled with a few dominant plant species.

The second technique this project will be using is Occlusion Culling, a technique that deactivates any mesh that is not visible to the camera. An example of occlusion culling is if the camera is close to a tree in the scene, any meshes behind that tree are deactivated, as the camera wouldn't be able to see them anyway, thus decreasing the amount of draw calls needed to render the scene (Figure 6).

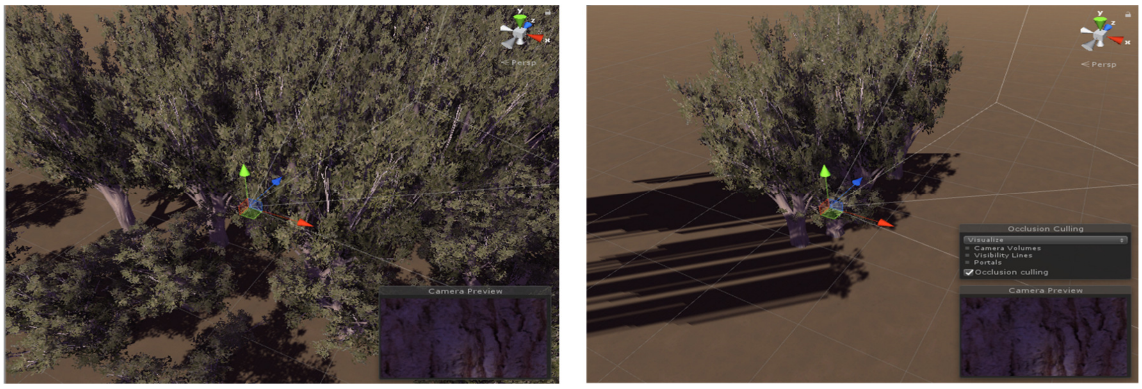

Figure 6. An example of occlusion culling when the camera is close to an object. Left: No occlusion culling. Right: Occlusion culling implemented.

GC optimization is the third technique for increasing the project's FPS. Garbage, in terms of VGEs, is the term used for memory that has been set aside to store data, but is no longer being used. GC is a process that cycles through the allocated memory and frees up unused memory for other tasks.

GC is an important process for keeping the project's memory usage at a manageable size but if it occurs too frequently it can seriously affect the project's FPS. To combat this, there are a few approaches to consider. One is to keep objects that need to be allocated to the heap at a minimum so whenever GC occurs, it happens quickly. Another approach is to reduce the frequency of object allocations and deallocations, thus reducing the frequency at which GC transpires. The final approach that can be taken is to manually invoke GC at predictable and convenient times.

3.3 Height Generation

Generating and customizing terrain heights in contemporary VGEs are usually done using heightmap textures. The VGE interprets the heightmap texture given to it to change the height of the terrain based on the colour of each pixel on the texture. To generate a completely new texture to feed into a VGE terrain (hereby referenced to as Terrain), one of the obvious options is a simplex noise texture. A simplex noise texture is a texture that consists of randomly placed black and white pixels. This however, will not create a realistic environment as it lacks any correlation between pixels that is needed to simulate gradual hills and valleys.

So, noise textures are the solution, but simplex noise is too cluttered and erratic to create a convincing environment. A solution to this is the use of a different type of noise; Perlin noise. Perlin noise is a type of ‘gradient noise’, created by Perlin in 1983.

3.4 Terrain Texture Generation

In this project, we want certain textures to be applied to the Terrain based on different ‘rules’ that we can define. These rules could be things like; “only place rock textures on steep cliff sides” or “only place snow textures on the tops of mountains”.

The first rule we need to implement is how to place the grass texture, this is important as forest environments have plenty of lush grass within them. We will be using both the height and steepness of each pixel in the Terrain heightmap to implement our ‘grass rule’, as we want grass to primarily grow on flat terrain at lower altitudes. The flatness portion of the rule is a bit complex, as the angle of each pixel in the terrain isn’t a ‘normalised’ value, normalised meaning it is not a value between 0 and 1. Another hurdle is feeding the pixel’s height into the algorithm in a way that allows for smooth transitions between grass being placed and grass being ignored. The algorithm we use to calculate this grass rule is as follows:

$$x = clamp(\cosh \llbracket g/h \rrbracket - 1.25) - clamp((s * s)/(t/f))$$

Where x is grass texture transparency from 0 – 1; g is grass height lenience; h is height of the current pixel; s is steepness of the current pixel; t is the max height of the Terrain; f is grass flatness lenience

This algorithm returns the value *x* as the ‘weight’ of the grass texture from 0 to 1, with 0 meaning grass with full transparency (thereby invisible) and 1 meaning grass with no transparency (fully visible). This 0 to 1 value is necessary for blending between different textures on the Terrain. The *g* variable in the algorithm is the grass height lenience; this variable controls the altitude at which grass can grow. The *f* variable pertains to how flat the pixel must be to be considered for grass placement. The lower the *f* variable, the steeper the pixel can be before being rejected by the grass rule.

After developing the grass rule into our Terrain Texture Generation system, the next rule to develop is the ‘rock’ rule. This rule will define where exactly rock textures should be placed on the Terrain to increase variation and detail. These rock textures should be placed on high and steep sections of the Terrain to simulate mountainous regions. Like the grass rule, we will be inputting both the height and angle of each pixel into our rock rule, except in this rule we will be using the 'normal' of each pixel to calculate its angle. A pixel's normal refers to its rotation value on all three axes (x, y and z). The algorithm used to determine whether the rock texture will be placed is as follows:

$$x = clamp((h/r\hat{}2\,) * \cosh\,⟦n/a⟧\,)$$

Where x is rock texture transparency from 0 – 1; h is the height of the current pixel; r is the rock height lenience; n is the current pixel's normal value on the z axis; a is the rock angle lenience

Like the grass rule algorithm, this algorithm returns the value *x* as the transparency of the rock texture from 0 to 1. The *r* variable refers to the rock height lenience and is used to make sure the rock texture is only placed at high altitudes. The *n* variable is the current pixel's normal value on the z axis which makes sure our rock texture is only placed on very steep inclines. Variable *a* refers to the rock angle lenience and is used to control how steep of an incline the pixel needs to be before it gets assigned as a rock texture.

3.5 Detail Generation

The next step in creating our immersive forest environment is detail generation. Details on a Terrain refer to small 'details' such as grass, flowers and smaller plants, thus they are a necessary addition. To generate these details in an optimal manner, we will be using the Terrain textures we generated in the last step. We can simply reference the texture on each pixel of the terrain and use that data to decide exactly what detail needs to be there.

For this texture referencing to work, the Detail Generation system needs to be able to calculate the alpha of each texture it samples, as the Terrain textures previously generated smoothly transition between each other. Once we retrieve this alpha value, it is a simple matter of comparing it against the alpha values of any other textures that could appear on that pixel. After getting the texture with the highest alpha value, the Detail Generation system can finally decide on what should be placed on that pixel.

3.6 Point Generation

After developing techniques to control the look of the Terrain itself, this section will be covering the placement of objects upon it. To do this, we will be using a technique that was investigated in the previous section, Poisson Disc Sampling. This technique essentially makes sure that each object is placed a certain distance away from any neighbouring object. By implementing this technique, we can achieve a random, yet uniform distribution of objects across our whole Terrain.

In this project, we want specific objects to be placed on specific parts of our Terrain. To do this we need to analyse the texture beneath each placed point before deciding whether it should be placed there or not. This selective placement will allow us to have better control over the look of the forest environment.

3.7 Shadow Map Generation

Now that we have developed the system that will place our initial objects throughout the scene, we can now start work on a type of context-sensitive object placement. The context we will be focusing on is light level, or how bright or dark each section of the Terrain is. This was ignored in the previous section as there was nothing in the scene to cast any shadows.

To properly evaluate the shadows within the scene, we must generate a *shadow map*. A shadow map is a black and white texture that illustrates both the size and position of all shadows in the scene. After generating this shadow map, we can then overlay it on top of the Terrain to find exactly where each shadow is.

Generating the shadow map takes place in two steps: Render Toggling and Render Texture Output. These two steps allow us to have direct control over exactly how and when the shadow map is generated. These steps, in conjunction with a single orthographic camera, allows us to visualize all the shadows in the scene.

The first step, Render Toggling, grabs all the objects placed through the Point Generation system and toggles the renderer of each object for one frame. This step is used to make sure the orthographic camera will only see the shadows casted by the objects and not the objects themselves. As this all happens in a single frame, it is not visible to the user as the project is running at 60 FPS.

The second step is Render Texture Output which controls the orthographic camera and its relevant outputs. This step takes place once all the renderers in the scene have been toggled off by the Render Toggling step. It turns the orthographic camera on and then assigns a custom ‘shadows only’ shader that makes the camera render in black and white, with black being shadows and white being no shadows. The camera then assigns what it renders to an output texture called a Render Texture.

After assigning the Render Texture, we need to convert it to a normal image texture for evaluation. This is a simple matter of reading and transferring all the colour data from the Render Texture to the new image texture. Once the colour data has been transferred to the image texture, we can clear the Render Texture, so it can be used in later shadow map passes.

3.8 Life Cycle Simulation

With the all the necessary objects placed, this section will be covering the simulation of their life cycle. This system will control how and when a plant in the virtual environment should grow. There will be two 'modes' for any object that is simulating their life cycle: *Stepped Aging* and *Smoothed Aging*.

Stepped Aging is the mode that takes the least amount of resources as it only updates the age of an object in steps. These steps can occur once every five to ten seconds and significantly improve the performance of the scene. Smoothed Aging is the opposite of Stepped Aging, it takes more resources but gives the user a better experience as they see the object smoothly age over time. This mode updates the objects age once every frame, giving it a very smooth transition between young and old.

To properly use both modes, the Life Cycle Simulation system will check all objects that are currently aging within a radius around the main camera. If an object is within the radius, it will use Smoothed Aging, thus improving user immersion. If an object is outside the radius, it will use Stepped Aging instead, thus improving the performance of the scene without impacting user immersion. These two modes are crucial in making sure this system is optimized for a real-time scene, as having hundreds of objects updating their age at once will have an adverse effect on the projects FPS.

3.9 Day/Night Simulation

The final system we will be implementing is the Day/Night Simulation system. This will allow for a more immersive user experience and will make the forest environment less static. The two main conditions this system needs to consider are the lighting and overall colour of the scene. For example, simply having the sun set without changing the colour of the atmosphere will not create a believable sunset.

Firstly, we will develop the technique for moving and rotating the celestial bodies, the sun and moon, in our scene. Some considerations must be put in place for the key times of day to be realistic. Examples like having midday lasting longer and casting a harsher light on the environment compared to sunrise and sunset need to be accounted for.

One way of calculating the exact angle the sun needs to be at is to calculate the *solar hour angle.* The solar hour angle is a way of telling the time of day by looking at the angle of the sun in comparison to the horizon. By reversing this technique, we can instead calculate the angle of the sun based on the current time of day.

This technique will require the use of *decimal hours*, which is a type of time structure. A decimal hour is the combination of the standard 24hr time structure with the addition of a decimal calculation for minutes. For example, 3:30am (03:30 in 24hr time) will convert into 3.5 and 3:15pm (15:15 in 24hr time) will convert into 15.25.

After calculating the wanted decimal hour, we can now discuss the algorithm for calculating the angle of the sun from the time of day:

$$s = (360/24) * (d - 12)$$

Where s is the solar hour angle; d is the current decimal hour

In the above algorithm, we have divided 360 degrees by 24 to calculate the angle of a single hour. We also minus our decimal hour by 12 as the angle of the sun at sunrise is -90 degrees, while its angle at sunset is +90 degrees. By taking our hour angle and applying it to our decimal time, we can accurately identify the angle at which the sun should be. After calculating the needed angle of the sun, we also need to calculate the angle of both sunset and sunrise. These two angles are needed as we must use them for specific things to occur at those times. To do calculate them both, we need to account for two variables, latitude and declination.

Latitude is needed to simulate a geographical position for the purposes of finding the horizon. Declination is like latitude, but it is used to simulate an astronomic position instead. While, generally, a point’s declination is almost always within 0.01 degrees of its latitude, it is still useful for calculating the exact angles of celestial bodies.

With both latitude and declination accounted for, we can now calculate the angle of both sunset and sunrise:

$$ss = \operatorname{acos} \llbracket -\tan d * \tan l \rrbracket$$

$$sr = -ss$$

Where ss is the sunset angle; sr is the sunrise angle; d is the current declination; l is the current latitude

The algorithm above calculates the angle of the sunset by finding the arc cosine of the negative tangent of our declination times the tangent of our latitude. Once we have retrieved the angle of our sunset, we can simply use its negative value to find the angle of our sunrise.

Chapter 4: PROTOTYPING, RESULTS, AND EVALUATION

In this chapter, we will be conducting a scenario covering the practical design and implementation of each system discussed in the previous chapter. The implementation of these systems will be addressed in the same order in which they are approached in Chapter 3. The practical implementation of our theoretical system design and algorithms requires integration into a VGE, in this scenario's case, the Unity engine.

4.1 Prototyping

In this section we will be covering the practical design of the seven systems covered in Chapter 3. This design process is the crucial step between theoretical design and practical implementation, as it devises a way to integrate our theoretical design and algorithms into an actual functioning prototype.

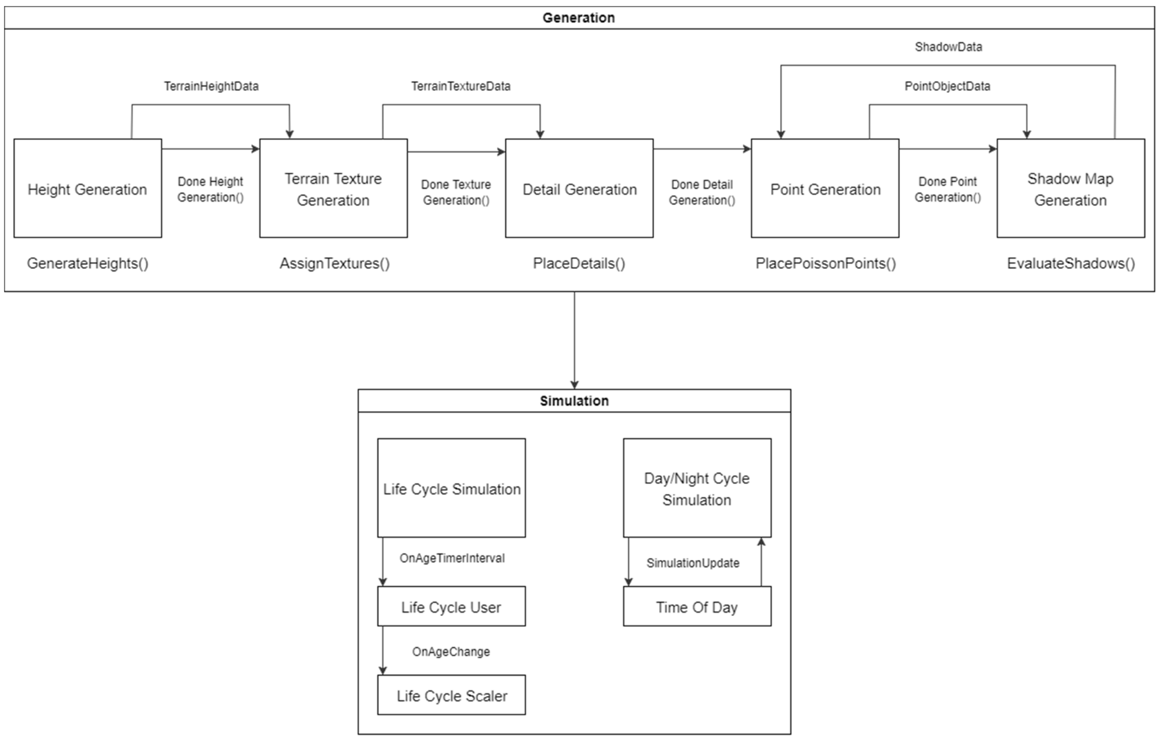

Figure 7 Overall Structural Diagram

As shown in Figure 7, an overall structural diagram of our system has been displayed. This diagram illustrates the order in which our systems will function, as well as the data to be passed between them. In addition, the methods used to complete each system's function have been laid beneath each system, and the methods used to continue the generation 'chain' have been displayed to the right of each system. The five generation steps must be completed before the two simulation systems can initiate. These steps include Height Generation, Terrain Texture Generation, Detail Generation, Point Generation and Shadow Map Generation. This multi-stepped design is crucial for guaranteeing each system will have the essential data they need before they perform their own generation. This multi-stepped design also allows us to easily modify how each system performs without influencing any other system. The simulation systems exist outside of our generation chain and will not affect any other system's functionality.

The first step, Height Generation, uses the 'GenerateHeights' method to generate a data type called 'TerrainHeightData' before passing it to the next step through the 'DoneHeightGeneration' method. Then, the Terrain Texture Generation step uses the 'TerrainHeightData' and generates a new type of data called 'TerrainTextureData' using the 'AssignTextures' method. It then passes that 'TerrainTextureData' to the next step using the 'DoneTextureGeneration' method. The third step, Detail Generation takes that 'TerrainTextureData' and generates its details in the 'PlaceDetails' method before continuing the generation chain with the 'DoneDetailGeneration' method.

The next step is Point Generation, which uses the 'PlacePoissonPoints' method to generate 'PointObjectData' that is then passed on to the next system, Shadow Map Generation. Shadow Map Generation takes the 'PointObjectData' and uses the 'EvaluateShadows' method to return 'ShadowData' back to the Point Generation system. Finally, the Point Generation system uses that 'ShadowData' to place objects in shadow before completing the generation 'chain'.

After the generation systems have been completed, the simulation systems can begin to run. The Life Cycle Simulation system runs in a top-down fashion, with the main Life Cycle system sending events down to each Life Cycle User and the Life Cycle Users sending events down to the Life Cycle Scalers. The Day/Night Cycle Simulation system performs its simulation in a cyclic manner, with the Day/Night Cycle system sending update events to the Time of Day system, and the Time of Day system sending data back into the Day/Night Cycle system.

4.1.1 Implementation

The implementation of all the aforementioned systems will give us an immersive virtual forest environment that is simulated in real-time. We will be implementing each system in the same order as they were discussed in the previous section.

The first system to implement is the Height Generation system. We first need to generate a perlin noise texture using Ken Perlin's perlin noise generation algorithm. After taking this perlin noise texture and implementing it into a VGE's terrain system, we can create more believable landscape features such as hills and valleys.

Because we are generating a heightmap when the program runs, we cannot feed the Terrain our heightmap texture directly beforehand. To get around this, we must iterate through each pixel on the texture we generate and assign their colour value to the Terrain's heightmap. We can continue this noise texture generation to create more visually distinct and diverse environments by layering these perlin noise textures over each other.

This layering of perlin noise textures is useful for adding more features to the Terrain. These features could range from giant mountains to shallow grasslands. To layer another perlin texture on top of our existing one, all we must do is add the colour data of each pixel in each texture together.

After generating the heights of our Terrain, the next step is to generate a texture to be placed onto it. This texture differs from the heightmap texture we generated previously as this texture will be the colour and literal 'texture' of the Terrain itself. Terrains in contemporary VGEs can be painted using multiple different textures, so they must be sectioned off from each other to allow blending between them. To do this, most VGEs partition each texture stored in the Terrain into separate layers. Combining the texture, its position on the Terrain and the layer it is placed on results in the Terrain variable, alphamaps.

Changing this alphamap variable requires consideration for all the values it holds; the texture itself, the position we want the texture to be placed and the layer on which the texture will be assigned to. To streamline this process as much as possible, we can assign most of these variables in the Terrain component itself. Both the texture and the layer value in alphamaps are both automatically assigned when we input the texture into the Terrains manual texture painting system. After inputting the texture into the Terrain component, all we need to do now is assign that texture to specific positions on the Terrain itself.

To retrieve the data that the previously developed grass and rock rules will be using, it is a simple matter of cycling through each pixel of the Terrain's height map and getting the height and steepness of each of them. We can use two functions to retrieve this data, GetHeight and GetSteepness. This GetHeight function returns the height of a specific pixel on the Terrain by supplying it with an *x* and *y* value and GetSteepness uses those same values while returning the angle of the pixel. By feeding all the relevant information into our grass (Figure 8-a) and rock (Figure 8-b) rules, we can give the Terrain colour and life-like textures.

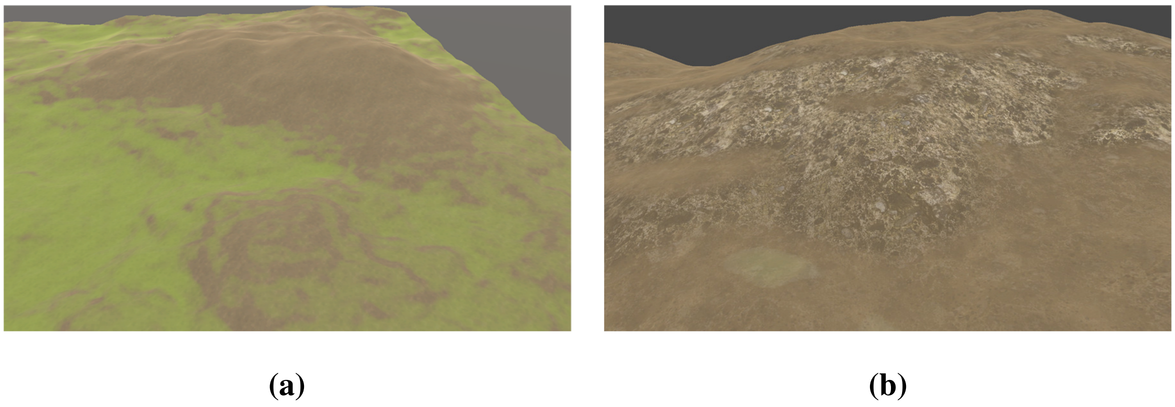
(a) **(b)**

Figure 8. Implementation of the grass rule and Implementation of the rock rule

With our Terrain Texture Generation system implemented, we now need to implement the Detail Generation system. To better optimise this system for real-time performance, we can implement a variable that controls the overall amount of details placed on the Terrain named 'DetailPlacementPower'. This variable will be used just before the Detail Generation system attempts to place a detail on the Terrain. If the detail's position in the Terrain's detail list is a power of DetailPlacementPower it can be placed, otherwise the Detail Generation system will skip over it (Figure 9).

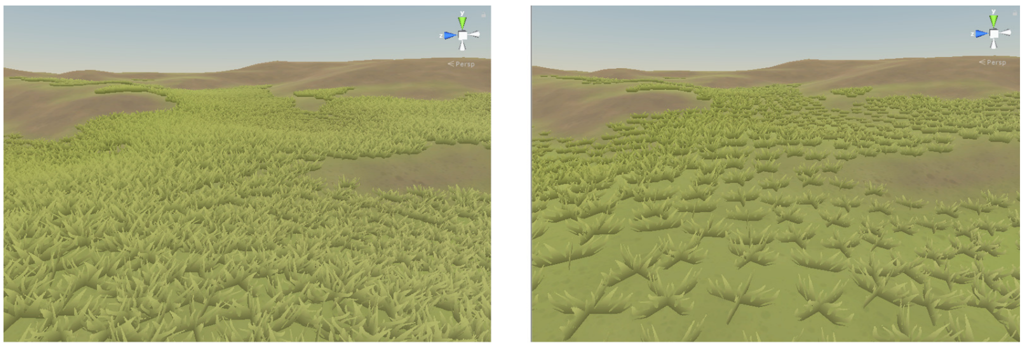

Figure 9. Left: DetailPlacementPower = 1. Right: DetailPlacementPower = 2.

We can also create a variable to do the opposite of our DetailPlacementPower variable and control exactly how many details should be placed per pixel on the Terrain. This variable is named ‘DetailsPerPixel’ and can be used to increase the density of our generated details at the cost of performance. The Detail Generation system uses this DetailsPerPixel variable when it decides a detail can be placed (Figure 10).

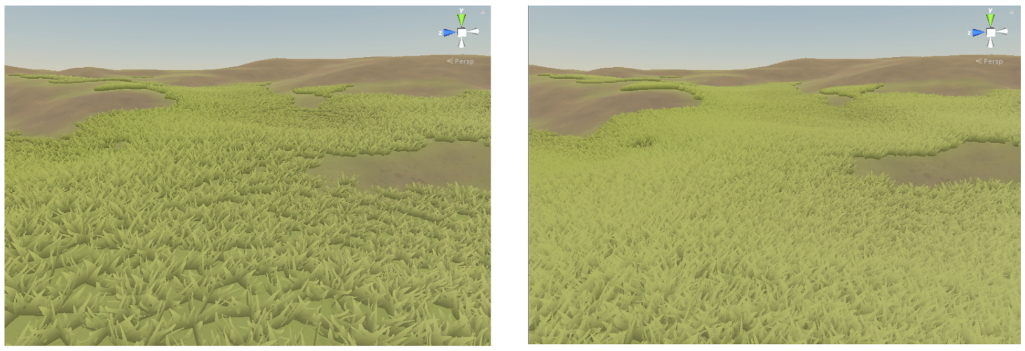

Figure 10. Left: DetailsPerPixel = 1. Right: DetailsPerPixel = 4.

Next up is the Point Generation system that implements Poisson Disc sampling to place points around our environment. By creating a class called T_Point to store the point’s *x, y* and *z* position values, we can accurately assign each point’s position on the Terrain. One issue with the Poisson Disc sampling technique is that it only calculates positions in a 2D space, meaning our objects will be placed at the same height regardless of the actual height of the Terrain. To solve this issue, we can use our previously developed GetHeight function and feed that value into our T_Point class.

With the basic functionality of the Poisson Disc sampling method implemented, we are now able to implement rules, like those used in the Texture Generation system, to change how these points should be placed. The first rule our Point Generation system will be implementing is a rule to determine where trees should be placed. We want this rule to only allow trees to be placed on dirt, allowing open grass plains to occur. To do this, we will be using the same method used in the Detail Generation system to sample the texture beneath each point (Figure 11-a).

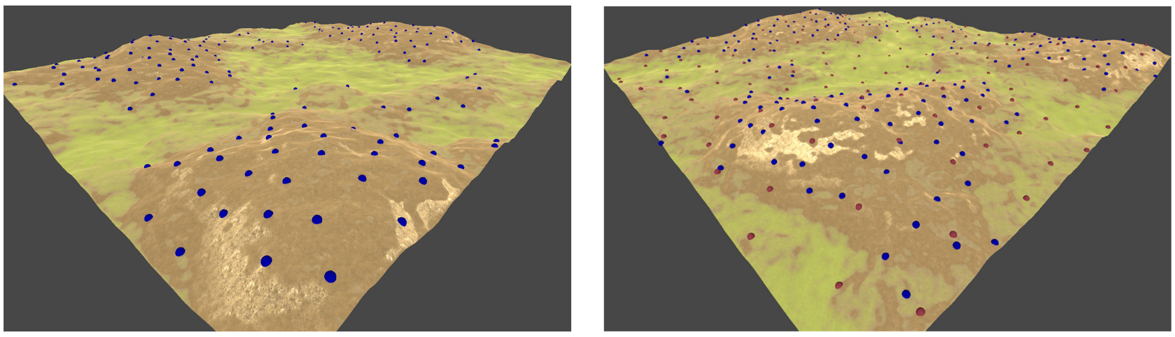
(a) **(b)**

Figure 11. Poisson Disc sampling using the tree rule and placing the rock points (Blue: Trees, Red: Rocks)

With these points placed via the tree rule, we now want to perform another Poisson Disc sampling pass to place rock points around the Terrain. Because of the way Poisson Disc sampling works, we need to separate these new rock points from the previously placed tree points. This is because we want the rocks to be able to be placed close to the trees on the Terrain and merging the rock points with our tree points will make sure they will stay a certain distance away. Because the rocks being placed are not very large or obstructive, we want them to be placed throughout the scene no matter the height, steepness or texture beneath them.

With our two sets of points placed throughout the scene, we can now replace them with actual objects. To do this, all we need is a reference to the object we want placed at any point. As we have already stored the position of these points, placing objects is a simple matter of iterating through them all and assigning the relevant object at their position.

After placing our objects around the Terrain, we can now move on to implementing the Shadow Map Generation system. This system will control how light level should affect the placement of different objects around the environment.

With the implementation of the Shadow Map Generation technique developed in Chapter 3, we are left with a shadow map texture that displays all the shadows in the scene. With this shadow map texture created, we can now evaluate and place objects around the terrain by injecting the shadow map texture into our Point Generation system. It works quite similarly to the tree rule within the Point Generation system, but instead of evaluating the textures on the Terrain, we will be evaluating the pixels on the shadow map.

With our contextual placement implemented, the next system to be implemented is the first simulation system, the Life Cycle Simulation system. This system controls the life cycle of all the trees and plants within the forest environment. To begin implementation, each of the trees placed by the Point Generation system will be assigned a component called a *Life Cycle(LC) User*. This component will control variables such as the speed at which each tree will age, how old it can be and how quickly it should reach maturity.

These LC Users will grow on their own, using the previously stated variables to simulate their own life cycles independently. However, when hundreds of LC Users are calculating their life cycles independently, it can have a significant adverse effect on the project's FPS. To alleviate the FPS strain, we can use a single manager class to control every LC User in the scene. This class will be called the *LC Manager* and will send out alerts and events whenever necessary to all LC Users in the scene. The LC Manager will send events to all the LC Users in the scene to tell them when to update their own life cycles. These events occur about once every five seconds, thus massively decreasing the amount of times the LC Users need to update. This technique of sending events to notify when the LC Users need to age is referred to as *Stepped Aging*.

With each tree's life cycle now simulated, it now needs to be visually apparent they are growing. This will be controlled by assigning a new component to each LC User, the *LC Scaler.* The LC Scaler is a child component to the LC User and is reliant on the life cycle variables within it. The LC Scaler has its own internal variables that control it's minimum and maximum size and the speed at which it will grow.

The final system that needs to be implemented is the Day/Night Simulation system. This system will simulate the movement of the two major celestial bodies, the sun and the moon. As we have already developed the algorithms needed to accurately calculate the sun's rotation at any time of day, all we need to do is simulate the time of day.

Simulating the time of day is simple; we just need to create variables for *seconds*, *minutes, hours* and *days.* By iterating our seconds value by 1 every second, we can accurately simulate time of day. Once that seconds variable reaches 60, we increase our minutes value by 1 and reset our seconds value to 0, then we repeat this step for minutes and hours.

With our time of day simulated, we can now covert it into decimal hours. The hour value of a decimal hour is simply the hour value in 24hr time, while the decimal value (minutes) requires some calculation. Finding the percentage of our minutes is as simple as dividing our current minutes value by 60, giving us a decimal value between 0 and 1.

There is one issue with just using minutes for our decimal hour however, as it will only update our decimal hour once per minute. This will result in small 'jumps' in our simulation as the sun's angle moves rigidly with the update of our decimal hour. To combat this, we need to incorporate our seconds value in our decimal hour as well.

To properly integrate our seconds value into our decimal hour, we need to divide it by 6000 and add it into our decimal hour. The reason for dividing it by such a high value is because we only want our seconds value to affect the decimal hour's third and fourth decimal place.

For example, if the time of day is 06:30 and our seconds value is at 35, our decimal hour value will be 6.5058. By doing this, we will now have a smooth transition between the minutes of our decimal hour. With our decimal hour fully implemented, we can now use the sun hour angle algorithm and calculate the angle of the sun. Before we use the sun angle we have calculated, we first need to set up how the sun should rotate. To do this, we can create a 'dummy' object in the centre of our Terrain and make the sun and moon children of it. By positioning our celestial bodies as children of this dummy object, we can simply rotate the dummy to simulate the spherical rotation of our sun and moon (Figure 12).

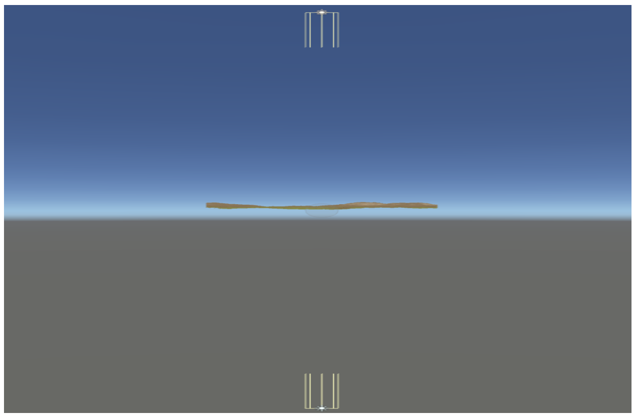

Figure 12. The position of the celestial bodies as children of the dummy object (Top: Sun, Bottom: Moon)

With our celestial bodies rotating, we now need to calculate the angles of our sunset and sunrise. In this project, we will be using the latitude of the Tropic of Capricorn and a declination value of 25 when calculating these angles. By using these values and applying them to our previously discussed sunset and sunrise algorithm we are left with 1.357 degrees for our sunset and -1.357 for our sunrise.

After finding these values, we can now control the overall look of both our sunset and sunrise. To do this, we will be using a gradient and evaluating it to find the colour we need. This gradient will be evaluated by using a 0 to 1 value, with 0 being the leftmost side of the gradient and 1 being the rightmost side of the gradient.

To retrieve this 0 to 1 value, we use the following formula:

$$x = (y + (i - r))/(j + s)$$

Where x is the gradient value from 0 to 1; y is the current sun hour angle; i is the minimum value for sunrise; j is the maximum value for sunset; r is the angle of sunrise; s is the angle of sunset

In the above algorithm, we have retrieved a value from 0 to 1 by evaluating our current sun hour angle. The value i in this case is 90, as sunrise occurs when the sun hour angle is -90 degrees. By taking our sunrise angle from i, we are left with the exact angle at which our sun will rise past the horizon. The value j in this case is 180 because our sunset will occur at +90 degrees. We then add the angle of our sunset to j to retrieve the exact angle at which our sun will set behind the horizon. We can now take the evaluated colour of our gradient and apply it to the sun to change the look of our environment at sunrise and sunset.

We can simply reverse the gradient algorithm and create a new gradient to simulate the colours of night, from twilight to midnight. Since night is generally shorter than day, we can calculate the leftover decimal hours from our sun angle algorithm to calculate this gradient colour. By doing this, we have now simulated both day and night, with the realistic position of celestial bodies, as well as the realistic colours present throughout.

4.2 Results

From the previous section, we have completely implemented the theoretical systems and algorithms designed in Chapter 3 into the Unity engine. During this process, we found that certain things needed to be considered when translating our theoretical work into a practical solution. Many of these considerations involved the Day/Night system, starting with the accurate simulation of time, from seconds to hours. They also included the need for a 'dummy' object when rotating the two celestial bodies in our Day/Night simulation system, as well as the need for evaluating a gradient texture for distinct colours to appear at various times of day.

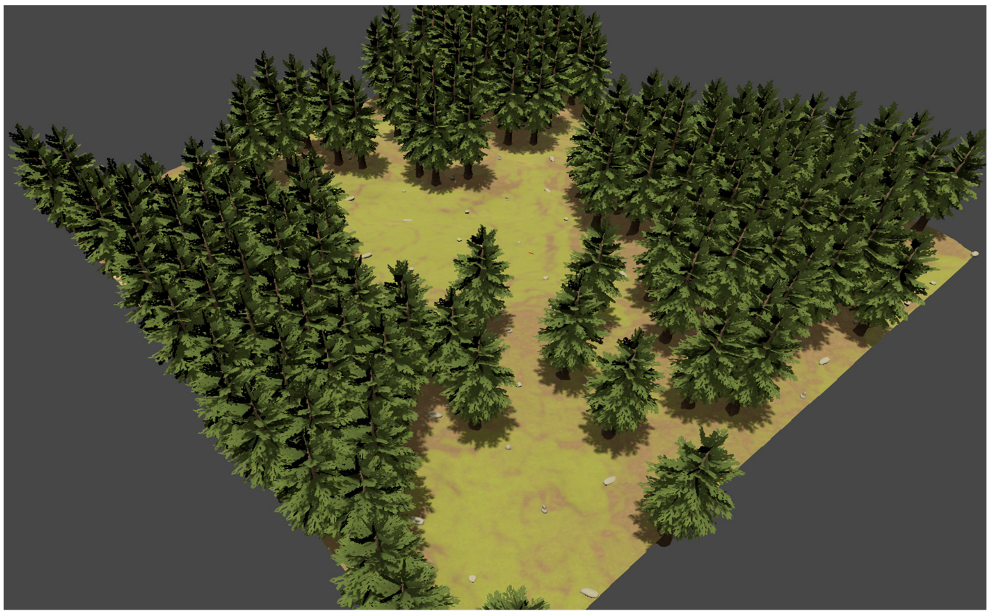

Figure 13 Placing objects on generated points

Figure 13 illustrates the placement of actual objects using the points generated by our Point Generation system. By designing the system to hold all the necessary data in a special class, we could simply use that data to place objects around our environment without worrying about unwanted clustering or object clipping. The implementation of our point placement rules also allowed us to easily create new rules for placing objects at specific positions, as seen in the following figure.

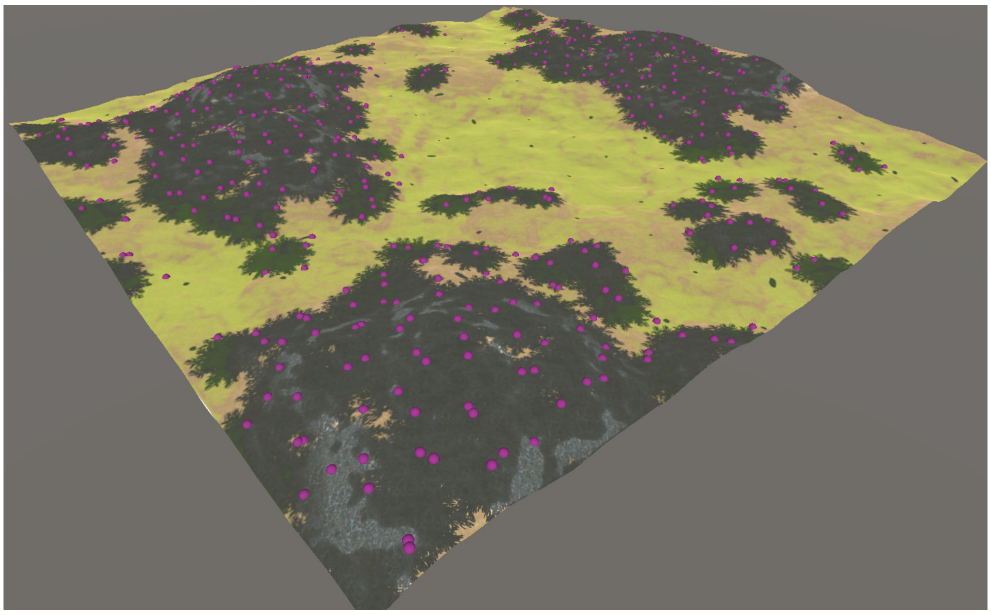

Figure 14 Using the Point Generation system to place points in shadow

In Figure 14, we can see how we were able to use the rules implemented in our Point Generation system to place points in specific areas, in this case shadows. By designing and implementing the Shadow Map system in a way that allowed it to easily communicate with the Point Generation system, we could use the functionality of both to place these points in shadow.

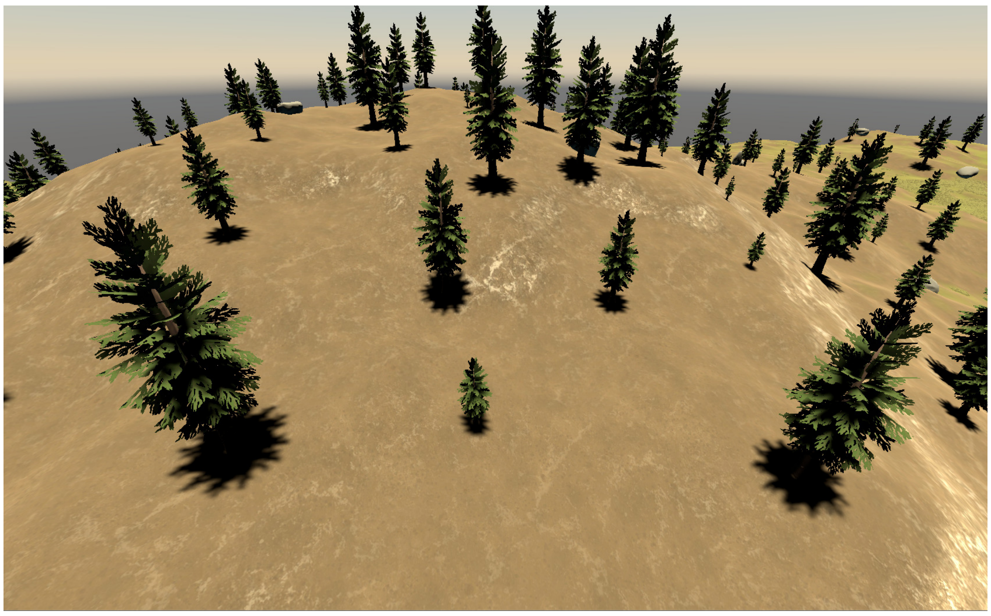

Figure 15. Trees growing using the LC Scaler component

Figure 15 showcases our Life Cycle simulation system running within the engine, with the trees placed around the terrain by our Point Generation system growing at random speeds. In addition, when these trees grew too old and died, they dropped one to two seeds in the area around them, thus keeping the environment alive and ever changing.

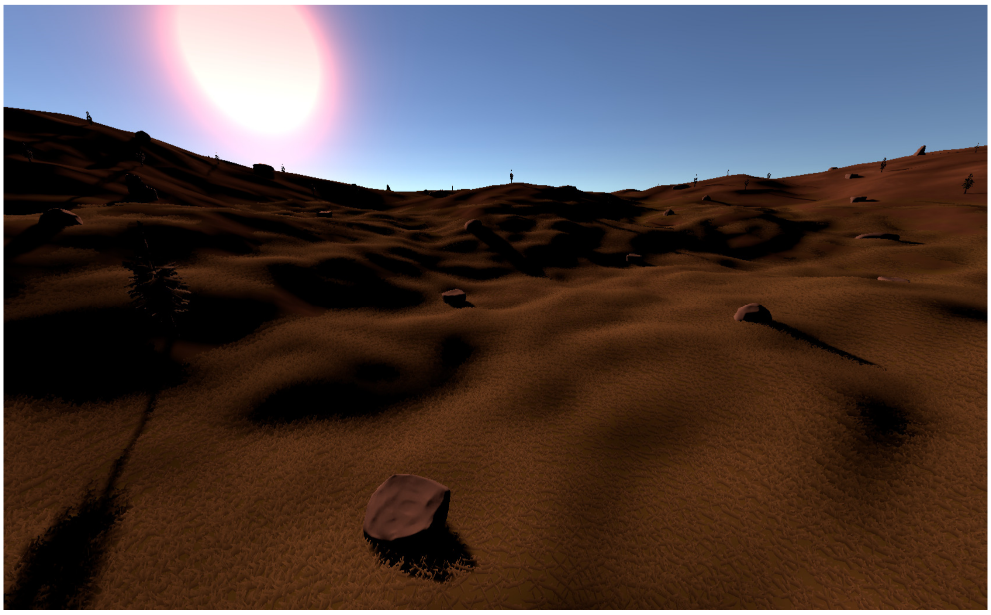

Figure 16 The colour of the environment at sunset.

Figure 16 displays the successful implementation of the Day/Night simulation system in the Unity engine, with the rotation of the celestial bodies being controlled by the aforementioned 'dummy' object, which in turn, is controlled by the algorithms designed in Chapter 3. In addition, it also showcases the successful evaluation and application of a colour gradient to the overall colour of the environment.

Our results are visible from the implementation of the various systems in the previous section. This is also illustrated by the figures shown in both the previous section and this one, with our generation and simulation systems fully integrated and implemented into the Unity engine. Furthermore, we were successful in the implementation of our generation 'chain' design, allowing us to work on each system in order, without the specific implementation of each system affecting the others.

4.3 Evaluation

With the design and implementation of our five generation systems and two simulation systems, we could successfully generate and simulate a realistic virtual forest environment. Each of our systems was successfully translated from theoretical concept to practical implementation. By documenting and analysing the implementation of each system, we were able to demonstrate its functionality and utility.

Also, our system was successfully integrated into another honours project undertaken by Chris Osmond in 2017, titled 'A Generic Architecture for an Ecosystem of Autonomous Artificial Animals using Dynamic Considerations'. This research outlined the design and implementation of an artificial ecosystem for artificial animals. We were successfully able to integrate this project into Osmond's by implementing new Poisson Disc Sampling passes into our Point Generation system. By doing this, we could generate food and water sources for the artificial animals to interact with, thus decreasing the amount of work Osmond needed to perform and increasing the immersion of his own project.

Chapter 5: CONCLUSION

In this chapter, we will be addressing our research contributions, our findings from this study, the answers to our research questions and the potential research problems exposed by our research.

5.1 Research Contribution

As has been established throughout this study, generating a virtual forest environment for real-time requires many in-depth considerations to achieve a realistic and immersive result. Through the investigation, development and implementation of seven unique systems, we have successfully generated a realistic virtual environment that is simulated in real-time.

From the generation of the geographical appearance of the environment, to the simulation of day and night, we have always taken optimizations for real-time into account. This has left us with a virtual forest environment that can be used in many different real-time applications, from video games to simulations (Figure 17).

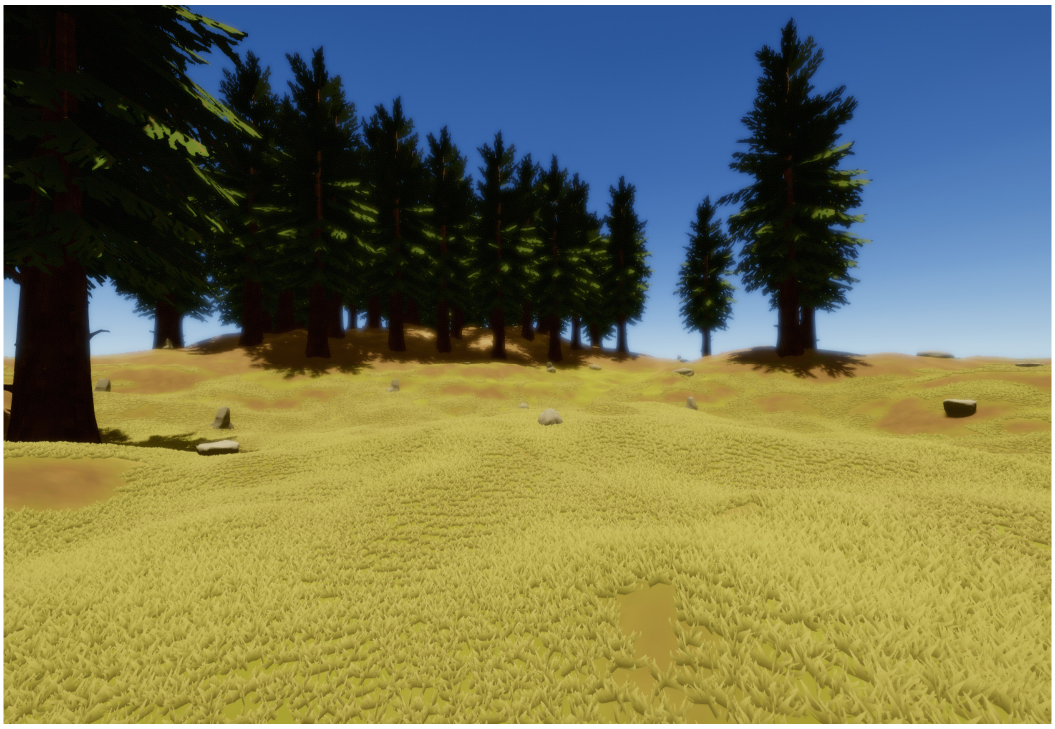

Figure 17. The generated virtual forest environment

5.2 Main Findings

We have investigated, designed, implemented and integrated seven systems which included Height Generation, Terrain Texture Generation, Detail Generation, Point Generation, Shadow Map Generation, Life Cycle Simulation and Day/Night Simulation. This integrated system allowed us to generate and simulate a realistic virtual forest environment in real-time. During this process, we found that these seven systems integrate and complement each other very well in terms of realism and user immersion. In addition, we found that incorporating optimization techniques into our system allowed the created artefact to efficiently render and simulate in real-time.

5.3 Answers to Research Questions

Through our research, we can now answer the three research questions stated in Chapter 1.

- Our first research question was "What are the required methods and features for generating a virtual forest environment using procedural content generation (PCG)?"

    - The answer to this question was discovered through the design and iteration of our seven proposed systems. The required PCG methods and features required for generating a virtual forest environment start with the generation, modification and texturing of the terrain in which the forest resides. The next feature needed is the random, yet controlled placement of trees and other objects throughout the generated terrain. This random placement needs to be controlled through specific rules to increase the diversity of the scene by creating random areas of interest, such as grass plains. In addition, this placement feature should evaluate the shadows cast by the environment before placing light-sensitive objects, thus increasing the environments visual diversity and realism. Finally, the simulation of the life cycle of various flora in the scene and cycle of day and night will improve the immersive aspect of the virtual forest environment by increasing its realism.

- The second research question we discussed was "How can different PCG systems be incorporated into a unique method of fully generating and simulating a virtual forest environment in real-time?"

    - This question was answered in Chapter 3 and 4, with our five PCG systems and our two simulation systems being designed and implemented into a singular, modular 'multi-system'. This multi-system was designed in a way that allows for the creation and integration of completely new systems, as well as the modification and iteration of the seven existing systems.

- The third and final research question was "How can the proposed system be analysed and evaluated using the Unity engine?"

    - This question was answered through our implementation scenario in Chapter 4, where we sequentially stepped through each system and implemented them into the Unity engine. As our systems and algorithms were designed to be implemented into a real-time engine from the beginning, integration into the Unity engine was a simple endeavour. The evaluation of our proposed system was exactly this implementation scenario, as it allowed us to see our system's robustness and utility when being implemented.

5.4 Future Work

Other research problems exposed by our development of this system include: (i) improvement on the realistic simulations by simulating rain-fall, seasons, moon cycles and wind; (ii) improvement on the geographical generation by generating rivers, lakes and ponds that would add more distinct features to the environment; and (iii) improvement on the placement system by considering the water content of the soil and the dominant plant species of the environment.

Reference List

Bridson, R. (2007). *Fast Poisson disk sampling in arbitrary dimensions.* ACM SIGGRAPH 2007 Sketches On - SIGGRAPH '07 http://dx.doi.org/10.1145/1278780.1278807

Ch'ng, E. (2011). *Realistic Placement of Plants for Virtual Environments.* IEEE Computer Graphics And Applications, 31(4), 66-77 http://dx.doi.org/10.1109/mcg.2010.42

Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., & Prusinkiewicz, P. (1998). *Realistic modeling and rendering of plant ecosystems*. Proceedings Of The 25th Annual Conference On Computer Graphics And Interactive Techniques - SIGGRAPH '98. http://dx.doi.org/10.1145/280814.280898

Ebeida, M., Mitchell, S., Patney, A., Davidson, A., & Owens, J. (2012*). A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions*. Computer Graphics Forum, 31(2pt4), 785-794. http://dx.doi.org/10.1111/j.1467-8659.2012.03059.x

Floyd, R., & Steinberg, L. (1976). *An adaptive algorithm for spatial grey scale*. Proceedings Of The Society Of Information Display, (17), 75–77

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). *Procedural content generation for games.* ACM Transactions On Multimedia Computing, Communications, And Applications, 9(1), 1-22 http://dx.doi.org/10.1145/2422956.2422957

Kailkhura, B., Thiagarajan, J., Bremer, P., & Varshney, P. (2016). *Stair blue noise sampling.* ACM Transactions On Graphics (TOG) - Proceedings Of ACM SIGGRAPH Asia 2016, 35(6)

Kailkhura, B., Thiagarajan, J., Bremer, P., & Varshney, P. (2016). *Theoretical guarantees for poisson disk sampling using pair correlation function.* 2016 IEEE International Conference On Acoustics, Speech And Signal Processing (ICASSP) http://dx.doi.org/10.1109/icassp.2016.7472145

Kelly, G. (2008). *An Interactive System for Procedural City Generation* (pp. 1 - 3). Blanchardstown: Institute of Technology Blanchardstown.

Lane, B., & Prusinkiewicz, P. (2002). *Generating Spatial Distributions for Multilevel Models of Plant Communities.* Proceedings Of Graphics Interface, 27-29 May 2002(Calgary, Alberta), 69–80. Retrieved from https://pdfs.semanticscholar.org/ef0e/f687f159099ebc5fda02cda4306846e48f3a.pdf

Li, H., Zhang, X., Meng, W., & Ge, L. (2017). *Visualization of Tomato Growth Based on Dry Matter Flow.* International Journal Of Computer Games Technology, 2017, 1-12 http://dx.doi.org/10.1155/2017/2302731

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). *Procedural modeling of buildings.* ACM Transactions On Graphics, 25(3), 614 – 620

Natural Resource Biometrics. (2014). *Hopkins' Index of aggregation.* oak.snr.missouri.edu. Retrieved 5 June 2017, from http://oak.snr.missouri.edu/nr3110/topics/hopkins.php

Park, C., Pan, J., & Manocha, D. (2014). *Poisson-RRT*. 2014 IEEE International Conference On Robotics And Automation (ICRA). http://dx.doi.org/10.1109/icra.2014.6907541

Park, C., Pan, J., & Manocha, D. (2017). *Parallel Motion Planning Using Poisson-Disk Sampling*. IEEE Transactions On Robotics, 33(2), 359-371 http://dx.doi.org/10.1109/tro.2016.2632160

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). *A Design Science Research Methodology for Information Systems Research.* Journal Of Management Information Systems, 24(3), 45-77. http://dx.doi.org/10.2753/mis0742-1222240302

Perlin, K. (1985). *An image synthesizer.* ACM SIGGRAPH Computer Graphics, 19(3), 287-296.

Suhartono, Hariadi, M., & Purnomo, M. (2012). *Integration of Artificial Neural Networks into an Lindenmayer System Based Plant Modeling Environment with Mathematica.* Studia Universitatis Vasile Goldis Seria Stiintele Vietii (Life Sciences Series), 22(3), 411-418

Yan, D., Guo, J., Wang, B., Zhang, X., & Wonka, P. (2015). *A Survey of Blue-Noise Sampling and Its Applications*. Journal Of Computer Science And Technology, 30(3), 439-452. http://dx.doi.org/10.1007/s11390-015-1535-0

Ying, X., Xin, S., Sun, Q., & He, Y. (2013). *An Intrinsic Algorithm for Parallel Poisson Disk Sampling on Arbitrary Surfaces.* IEEE Transactions On Visualization And Computer Graphics, 19(9), 1425-143 http://dx.doi.org/10.1109/tvcg.