BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENERATING TERRAIN WITH ANIMATED WATER SURFACE IN UNITY

GENEROVÁNÍ TERÉNU V UNITY S ANIMOVANOU VODNÍ HLADINOU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**BORIS KARAVASILEV**
SUPERVISOR **VEDOUCÍ PRÁCE**	**Ing. MICHAL ŠPANĚL, Ph.D.**

BRNO 2021

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Bachelor's Thesis Specification

24089

Student:	**Karavasilev Boris**
Programme:	Information Technology
Title:	**Generating Terrain with Animated Water Surface in Unity**
Category:	Computer Graphics

Assignment:

1. Study the topic of representation and procedural generation of polygonal meshes.
2. Get familiar with state-of-the-art methods of terrain generation in computer games and with the game engine Unity.
3. Choose appropriate methods and design a solution for generating polygonal terrain models with diverse landscapes in Unity. Fill the lower regions of the landscape with a representation of water.
4. Propose a solution for visualizing the terrain in Unity including water surface animation.
5. Implement the proposed solutions.
6. Experiment with your implementation and optionally propose modifications of the used methods.
7. Perform experiments and discuss the results and possibilities of future improvements.
8. Create a short video presenting your work, its goals and results.

Recommended literature:

- According to the supervisor's instructions.

Requirements for the first semester:

- The first three items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Španěl Michal, Ing., Ph.D.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 12, 2021
Approval date:	November 12, 2020

Abstract

Within this thesis a procedural terrain generator and a water surface shader was designed. These components were used in the implementation of a generator of infinite worlds composed of islands in the sea. Traditional methods to world partitioning and object placement were modified to best suite this purpose. Additionally, an approach for generating terrain of islands with diverse shapes by pseudo-randomly generating points defining the terrain's properties was devised. The implemented island generator running in real-time is capable of challenging the existing similar tools for Unity.

Abstrakt

V rámci této práce byl navržen procedurální generátor terénu a shader pro vizualizaci vodní hladiny. Tyto komponenty byly použity k implementaci generátoru nekonečných světů složených z ostrovů rozesetých v moři. Pro tento účel byly modifikovány tradiční metody rozdělení světa a umísťování objektů. Dále byl navržen přístup ke generování terénu ostrovů s rozmanitými tvary pomocí pseudonáhodného generování bodů definujících vlastnosti terénu. Implementovaný generátor ostrovů běží v reálném čase a je výzvou pro podobné existující nástroje dostupné pro Unity.

Keywords

procedural island generation, procedural terrain generation, water surface shader, Unity

Klíčová slova

procedurální generování ostrovů, procedurální generování terénu, shader vodní hladiny, Unity

Reference

KARAVASILEV, Boris. *Generating Terrain with Animated Water Surface in Unity.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Španěl, Ph.D.

Rozšířený abstrakt

Při tvorbě video her je potřeba vytvořit velké množství různorodého obsahu což je pro mnoho malých herních studií a nezávislých vývojářů téměř nemožné zvládnout bez použití pomocných nástrojů. Jedna z velkých částí obsahu her založených na pohybu ve fiktivním světě je vytvoření samotného terénu. Časová náročnost na vytvoření takového terénu není malá a roste s velikostí světa. Proto jsou pro herní vývojáře velkou pomocí nástroje, které tento proces zjednodušují, či kompletně řeší. Po prozkoumání obchodu s doplňky pro herní engine Unity zvaného „Unity Asset Store“ se ukázalo, že existuje vícero komplexních nástrojů pro generování realisticky vypadající krajiny. Na druhou stranu, nástrojů zaměřených na generování ostrovů v moři se zjednodušeným, spíše uměleckým vzhledem schopných běžet v reálném čase i na mobilních zařízeních nebylo nalezeno mnoho.

Cílem této práce je navrhnout a implementovat nástroj pro procedurální generování terénu s animovanou vodní hladinou pro herní engine Unity. Tento cíl byl z výše uvedených důvodů konkrétněji zaměřen na generování ostrovů rozesetých v moři. Nejdříve byly prozkoumány existující řešení a byla vytvořena detailní specifikace požadovaného řešení. Následně bylo po rešerši použitelných metod navrženo řešení generátoru ostrovů a shaderu pro animování vodní hladiny. Dále byly obě části implementovány, otestovány a byla vytvořena ukázková aplikace demonstrující možnosti vzniklého nástroje.

Pro generování ostrovů byly modifikovány některé tradiční přístupy k rozdělení světa a metoda pro umísťování objektů na zašuměnou mřížku. Dále byla navržena vlastní metoda pro generování terénu. Tato metoda generuje body tzv. „terrain nodes“ jež ovlivňují charakter i typy terénu v jejich okolí. Díky těmto modifikacím a nově navrženému přístupu ke generování terénu je možné procedurálně generovat svět s různorodými ostrovy rozdílných typů, tvarů, velikostí i s různými umístěnými objekty na nich. Proto, aby aplikace využívající vytvořený generátor světa běžely plynule, byl navržen vlastní systém rozdělující výpočetní kroky rovnoměrně v čase. Prostor mezi vygenerovanými ostrovy je vyplněný animovanou mořskou hladinou realizovanou pomocí shaderu složeného z kombinace několika efektů. Prvním efektem je barva a průhlednost vody závislá na hloubce. Další efekt vizualizuje mořskou pěnu kolem pobřeží a objektů částečně ponořených do moře. Následně je přidán efekt mořských vln a na konec distorzi objektů pod vodní hladinou.

Výsledkem této práce je řada znovupoužitelných komponent, které dohromady tvoří procedurální generátor světa složeného z ostrovů obklopených animovanou mořskou hladinou. Generovaný svět je teoreticky nekonečný a je generovaný v reálném čase. Byla vytvořena webová i desktopová aplikace prezentující ukázkový svět generovaný vytvořeným generátorem. Obě tyto aplikace jsou prezentovány na přiloženém videu.

Vytvořený generátor má mnoho přizpůsobitelných parametrů a může konkurovat podobným již existujícím nástrojům dostupných pro Unity. Obě verze ukázkové aplikace byly testovány a běželi průměrně na 60 snímků za sekundu na referenčním stroji. Aplikace běžela plynule, ale větší ostrovy se nestíhali generovat před vstupem na hráčovu obrazovku. Z tohoto důvodu byla věnována pozornost sekci zaměřené na možnou paralelizaci některých úkonů v budoucnu. Po optimalizaci rychlosti generování ostrovů a přidání dalšího typu šumu pro generování strmějšího terénu ve skalních oblastech bych chtěl dát nástroj k dispozici dalším uživatelům Unity.

Generating Terrain with Animated Water Surface in Unity

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Michal Španěl, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

<div align="right">

....
Boris Karavasilev
May 10, 2021

</div>

Acknowledgements

I would like to thank my supervisor Ing. Michal Španěl, Ph.D. for his guidance, constructive feedback and words of encouragement when I needed them.

Contents

Chapter 1

Introduction

The goal of this thesis is to design and implement a procedural terrain generator, as well as a shader for visualising an animated water surface. Specifically, with the aim to provide a theoretically infinite world for a game where the player will be sailing across the sea on a boat, and visiting different islands. This will be implemented as an *asset* (reusable component) for the Unity game engine, that I could publish on Unity's market place with software tools called the *Unity Asset Store*.

After searching the *Unity Asset Store*, I found out that there are a lot of existing tools for generating realistically looking landscapes that would be difficult to compete with. Therefore I have decided to develop a tool focused on the niche games taking place mainly on a sea and with a non-realistic, rather a cartoon stylised look. This could make my tool interesting for other developers.

In the process of creating a video game, from now on referred to as a *game*, a major part of the development process is creating the content. Examples of game content are maps, levels, game rules, textures, stories, items, music, weapons, vehicles, etc. Small teams, individual developers and beginners in the field of game development often use different tools that help them to create the content for their games, because they cannot cover the full spectrum of skills that are needed for the creation of a complete game. Creating a tool capable of generating such a major part of the content as a terrain for their game could really help them.

When I was working on my first games, another problem besides the big task of terrain creation was a lack of knowledge about shaders and computer graphics. Because it is not a beginners topic, new shaders for animating water surfaces are always welcome by the community. The tool that will be created as a result of this thesis should help small teams with both tasks, generating terrain and animating a water surface by a shader.

Examples of different types of existing tools present on the *Unity Asset Store* are listed in the chapter 2. The main theoretical concepts underlying the implementation of the terrain generator and the animated water surface shader are described in the respective order in the chapters 3 and 4. The conceptual design of my terrain generator is explained in chapter 5. Implementation of the proposed design as a tool for Unity can be found in chapter 6. Chapter 7 summarises the goals of this thesis and their fulfilment, presents my results and possibilities for further development.

Chapter 2

Existing Terrain Generators for Unity

The results of the research done on the existing tools for procedural island generation available on the *Unity Asset Store* at the time of writing this thesis are presented in this chapter. Since the developed generator was intentionally aimed at a niche category there were found only three direct competitors all having fewer features than the implemented solution. On the other hand, there are multiple much more superior general-purpose world generators available, namely MapMagic World Generator[1], World Creator Professional[2] and others. They are not seen as direct competitors because they are not focused on generating islands in the sea, are big and complex, and are targeted at realistic terrains.

2.1 Island Generators

There were found only three other island generators available at the *Unity Asset Store*. Their implementation, features, price and user reviews are analysed in this section.

Procedural Island Generator

The tool named *Procedural Island Generator* is the most similar tool that was found [19]. Its price was $14.99 and had zero reviews. It has been evaluated based on the description, video and images provided on its page. The date of the last release was in November 2019 which is more than one year and a half ago, so it does not seem that the tool is being actively developed anymore.

This tool seems to be based on a different concept where it places objects on top of each other to create a terrain of an island. Since it does not generate a mesh for the terrain it is not smooth, but composed out of multiple ground representing objects that are visually distinguishable. On top of these objects that represent the ground are placed other objects like trees and stones.

According to the description, it can generate worlds of infinite or limited size. In the demonstration video and the provided images are presented two types of islands of similar style. The sea is represented by a plane of a single colour and an animated shader does

[1]https://assetstore.unity.com/packages/tools/terrain/mapmagic-world-generator-56762
[2]https://assetstore.unity.com/packages/tools/terrain/world-creator-professional-55073

not seem to be included in the package. Both the islands and the sea are visible in the screenshots in figure 2.1.

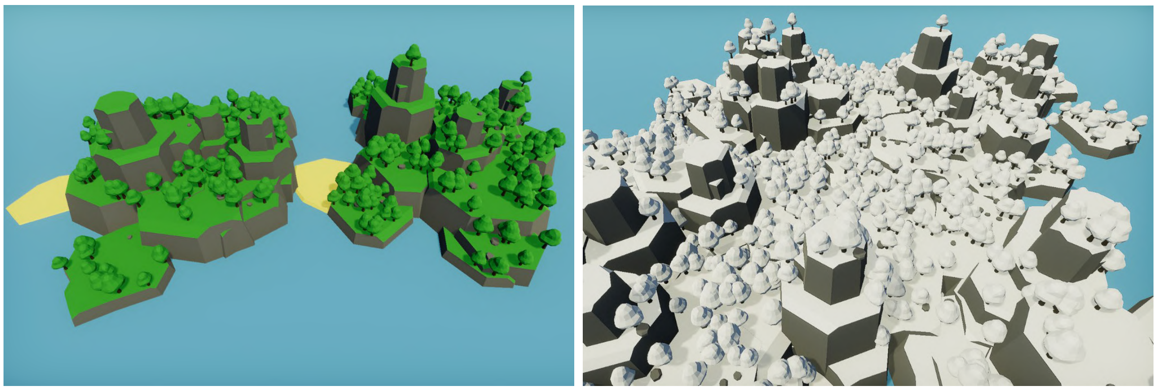


Figure 2.1: Screenshots of the generated islands, adopted from [19]

Procedural Island Basis Generator

The second found tool named the *Procedural Island Basis Generator* was free and had 15 ratings out of which 10 had reviews attached to them [18]. The overall rating was 5 out of 5 stars and the user reviews often praised the tool's small size and how easy it was to use. In contrast to the first analysed tool, this one had a video showing not only the results but also a quick guide on how to use the tool and its abilities. The last release of the tool was in December 2018 and also does not seem to be further developed since then.

The advantage of this tool is that it generates a terrain compatible with Unity's built-in system for creating terrains. That gives the users an easy way to edit the generated islands. This tool supports generating a terrain of a fixed size and does not include the water surface shader shown in the video. The tool at first generates terrain with sharp edges by the use of a cellular automaton as seen in the figure 2.2a and afterwards smooths out the terrain and applies Perlin noise to it as seen in the figure 2.2b. This tool does not generate a terrain texture nor supports the automatic placement of objects. It serves only as a generator of a base on top of which the user can create an island using Unity's built-in tools.

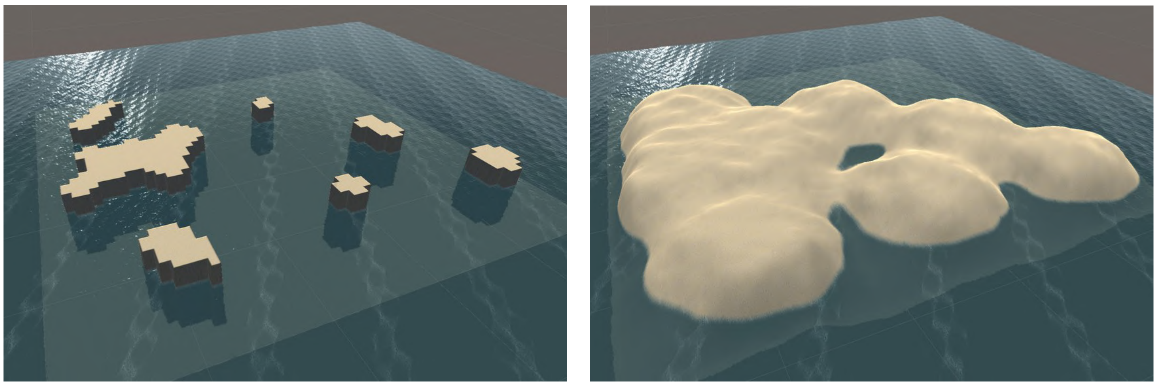
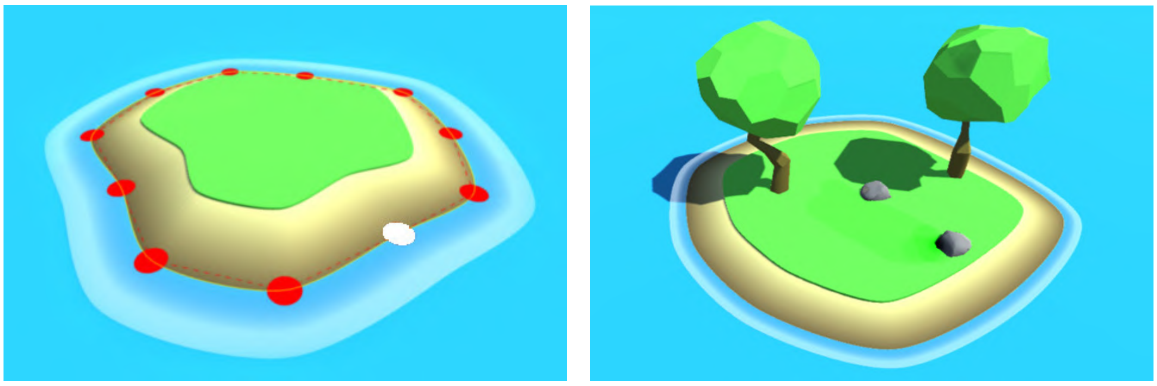

(a) Basis generated by cellular automaton.   (b) Basis with added smoothing and noise.

Figure 2.2: Screenshots of the generated island basis, adopted from [18].

Simple Bezier Island Generator

The last analysed island generator was the *Simple Bezier Island Generator* [20]. It was priced at $6.99 and had one five star rating along with a review complimenting the tool's documentation and possibilities of customisation. The last release date was in March, 2018.

This tool is focused on generating individual islands in yet another way. The adjustable border of an island is generated and the space inside of it is filled by the terrain of several levels with constant height. The edges of the islands are slanted and coloured with a gradient resembling a shore. It supports the random placement of objects. The sea is visualised as a plane of a single colour with a gradient resembling seafoam around the objects partially submerged in the sea. An example of a generated island is shown in the figure 2.3b and the adjustable border is visible in the figure 2.3a.



(a) Island's adjustable border. (b) Island with randomly placed objects.

Figure 2.3: Screenshots of the generated islands, adopted from [20].

2.2 General Purpose Terrain Generators

One of the advanced general-purpose infinite terrain generators available for Unity is the *MapMagic World Generator* [13]. It costs $112.50, has 279 user ratings, and 5 out of 5 stars rating. It supports dynamic generation of infinite realistically looking worlds as seen in the figure 2.4. It can generate the terrain's mesh, texture and also place objects on it.

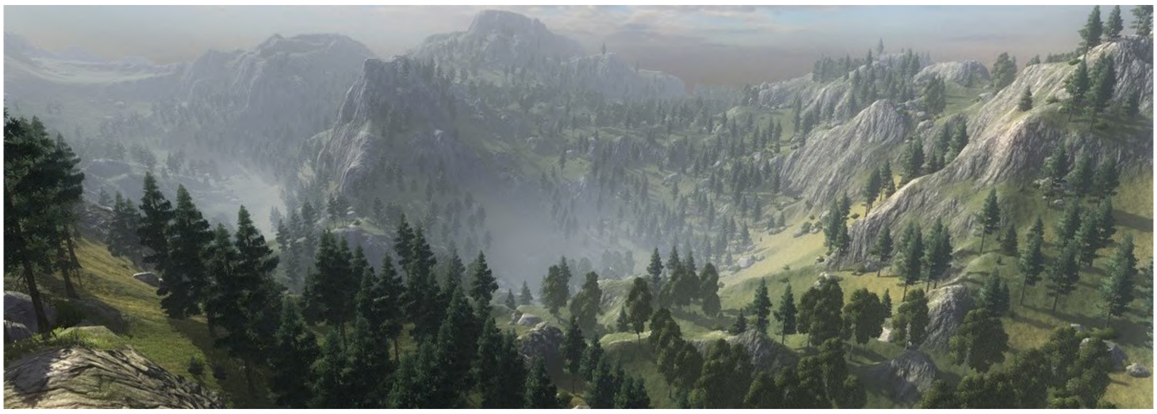


Figure 2.4: World generated by the *MapMagic World Generator*, adopted from [13]

Chapter 3

Procedural Terrain Generation

The theoretical concepts used in the design of the procedural terrain generator from section 5.3 are described in this chapter. Procedural terrain generation (PTG) is a type of procedural content generation (PCG). This thesis uses the definition of PCG being an *algorithmic creation of game content with limited or indirect user input* [3].

3.1 World Partitioning

Infinite procedurally generated worlds in games have to be divided into computable parts of finite size. These parts commonly have a square shape and a constant size. Each of these square tiles represents an area where a polygonal mesh visualising a chunk of terrain gets generated. When the generated mesh is intersected by a water surface plane it can represent islands in the sea. This common approach causes the islands and the sea bed to be connected and represented by a single mesh as seen in the figure 3.1.

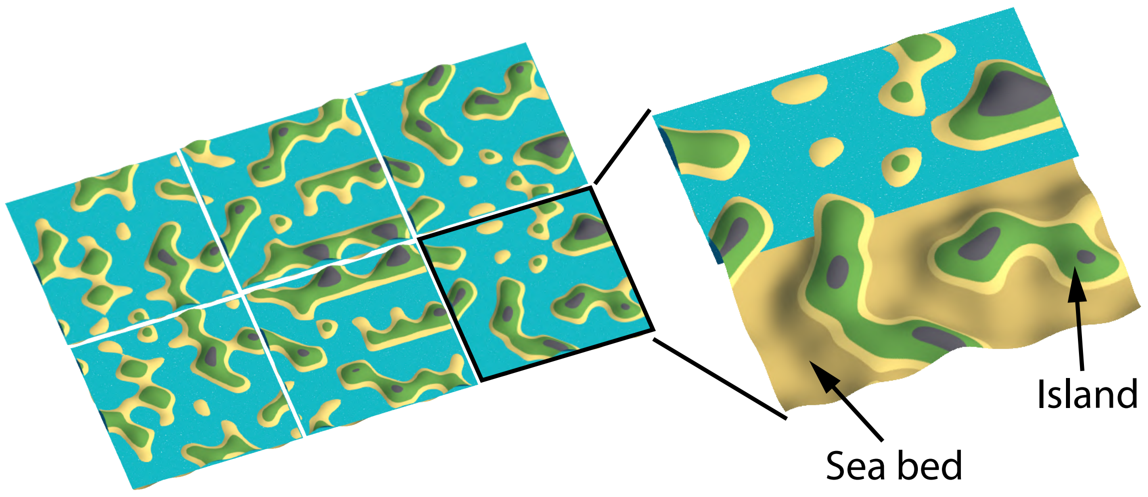


Figure 3.1: Common approach of representing the islands and the sea bed by a single mesh.

The world was partitioned this way in the initial prototype, but afterwards, it was modified by splitting the islands and the sea bed into independent objects as described in the section 5.3. The modification was made because of the limitations in the heightmap generation and the object placement challenges explained in the following two sections.

3.2 Terrain Chunk Generation

The traditional approach to generating terrain is from a heightmap. Alternative techniques such as voxel terrain representation are not discussed within the scope of this thesis as they are not suitable for the targeted type of games.

The basic approach is generating the height maps of all the terrain chunks from slices of a single continuous noise (e.g. Perlin noise or Rigid fractal noise). Inputs for generating a slice of noise for the specific chunk being the world coordinates of the chunk and the scale of the noise that is constant for all of the chunks as seen in figure 3.2. A combination of multiple octaves of noise (noise with different scale) into a more complex noise is also commonly used.

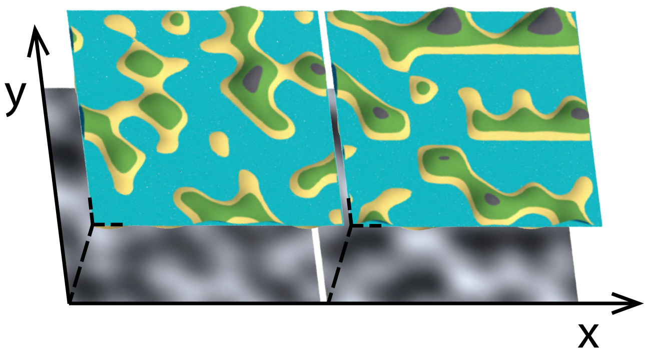


Figure 3.2: Heightmaps from slices of noise based on world coordinates and noise scale.

This method neither allows for generating the heightmaps from different types of noise at different parts of the terrain nor the control over the spacing of elevated areas that can represent islands. In the figure 3.3 it is seen how changing the scale of an example Perlin noise affects the size of all elevated areas representing islands. The interconnection makes it impossible to spread them further apart while keeping the same height of the water surface.

The two mentioned limitations result in a generated world with homogeneous islands that are too close to each other and lack areas of open sea, not very natural. These disadvantages have been solved by splitting the islands into independent objects and generating their heightmaps from multiple types of noise blended together as described in detail in the section 5.3. This separation also allows a precise control over the positions of the islands.

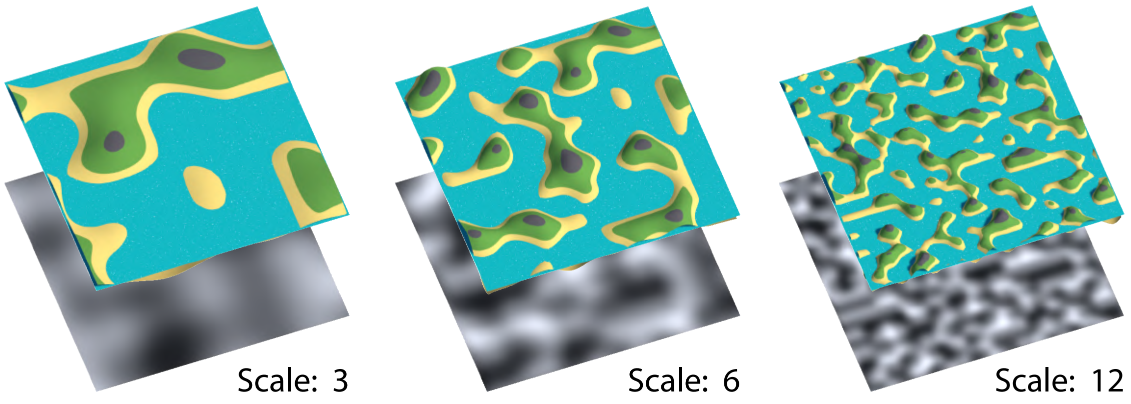


Figure 3.3: Homogeneous islands without areas of open sea.

3.3 Object Placement

There are two popular types of methods used for placing objects on a terrain. The first type are methods taking into consideration the positions of the previously placed objects to keep a minimum distance between them (e.g. Poisson disc sampling). The second type are methods that inherently guarantee a minimum distance between the objects by placing them on some kind of grid (e.g. Jittered grid) [4].

Because of the decision to split the world into islands of a small size being generated at once, it is possible to use both of the mentioned types of object placement methods. In case that the islands would need to be bigger, they would probably need to be split into multiple chunks to maintain a playable frame rate and low generation times (in order to generate the chunks before they appear in the field of view of the player).

In the proposed design a Jittered grid was chosen for both the placement of the islands in the sea and the objects on the islands themselves. This decision was made because this method would also be usable if the islands get split into chunks in the future without additional compensations. A jittered grid is also computationally faster compared to the Poisson disc sampling which tries to find suitable positions in a Monte Carlo way.

Otherwise if the first type of methods would be chosen, solutions to the following problems would need to be found. One of the problems with algorithms that are dependent on the already placed objects is the possibility of different outcomes when the terrain chunks get generated in different order. A simple example is a case where the algorithm has to place a tree on a small island with room for only one tree. The tree will be placed on that half of the island that gets generated first, for example by being closer to the player's position. When the player would approach the island from two different directions the tree would get placed on two different parts of the island, as illustrated in the figure 3.4.

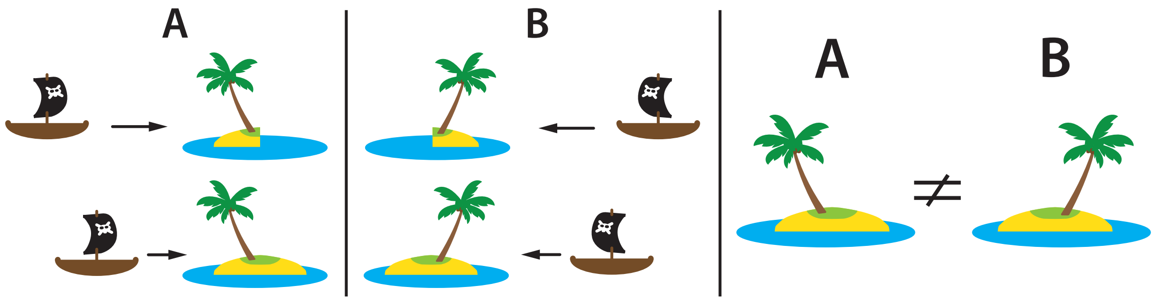


Figure 3.4: Dependence on the order of generation of Poisson sampling type methods.

This problem can be solved by saving data for each terrain chunk that has ever been generated. The saved data would grow linearly with the amount of generated terrain chunks and make the solution not purely procedural by default, which might not be wanted.

Another problem with the first type of methods is the possibility of unwanted visual artefacts such as colliding objects near the edges of two neighbouring terrain chunks. This could be resolved by a mechanism of exchanging data about placed objects near the edges between the terrain chunks or by allowing placing objects no closer to the edge than half of the minimum spacing between them.

3.4 Perlin Noise

Perlin noise is a type of gradient noise commonly used in the field of procedural generation. In this thesis it is used for generating heightmaps which are 2D grids where each of the cells represents the elevation of terrain at that location. It is also used by the jittered grid.

This thesis uses the default implementation built into Unity [15]. It allows sampling a noise value at arbitrary X and Y coordinates on an infinite 2D plane (limited by the range of floating point coordinates) and the returned values are between 0 and 1.

Perlin noise was developed by Ken Perlin and first published in a paper in 1985 [1] and later improved in 2002 [2]. Perlin noise is a type of gradient noise. That means that in order to generate its value a pseudo-random gradient has to be assigned to regularly spaced points in space. The value of the noise is calculated by interpolating between those points with the use of a smooth function. The following explanations are adopted from [6].

To generate Perlin noise in one dimension a pseudo-random gradient (or slope) is associated with each integer coordinate. The function value at the integer coordinates is set to zero. The value of a point somewhere between two integer coordinates is a result of interpolation between two values, namely the results of extrapolation of the closest linear slopes from the left and from the right to the point in question, depicted in 3.5.

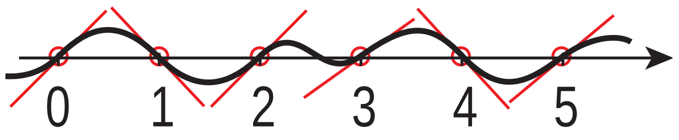


Figure 3.5: Calculation of Perlin noise value in one dimension, adopted from [6].

The interpolation is not linear with distance, because that would not satisfy the constraint of the noise function for its derivative to be continuous also at the integer points. Instead, a blending function with zero derivative at its endpoints is used. In Ken Perlin's original paper he used the Hermite blending function $f(t) = 3t^2 - 2t^3$. In his later paper from 2002 he replaced this function with a fifth degree polynomial $f(t) = 6t^5 - 15t^4 + 10t^3$ because even its second derivative is continuous which is desirable. Both depicted in 3.6.

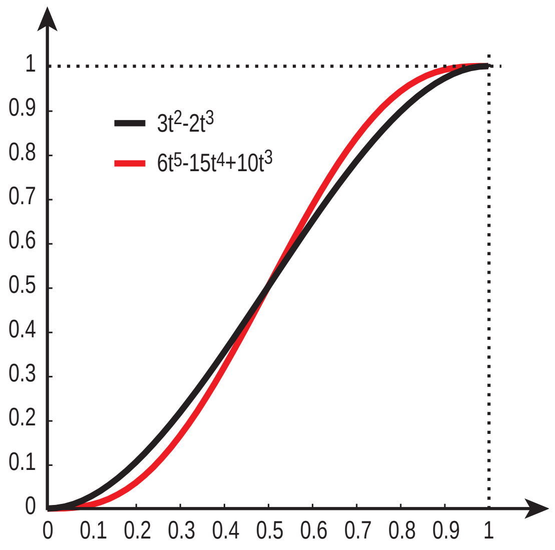
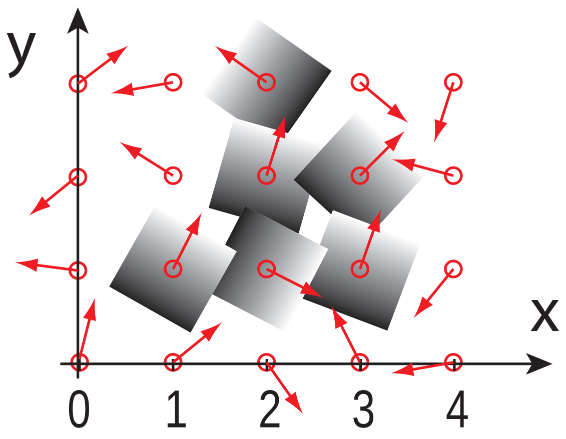
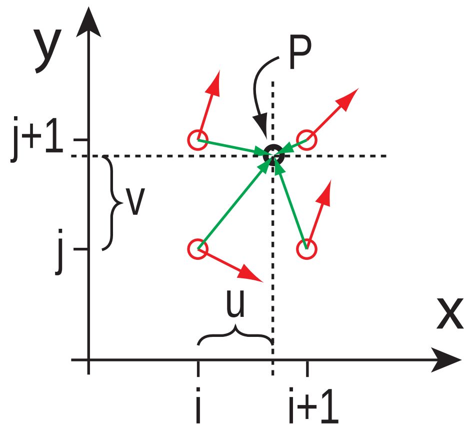


Figure 3.6: Comparison of the old and new (red) interpolation functions, adopted from [6].

To generate Perlin noise in two dimensions, pseudo-random gradient vectors of unit length are assigned to integer coordinate points that form a square grid as seen in the figure 3.7a. The value of the noise at a given point P is calculated by interpolating between scalar values of the closest four grid points. The scalar value for each grid point is calculated as the dot product of the gradient vector assigned to it (marked red) and the offset vector (marked green) going from the grid point to the point P, illustrated in the figure 3.7b.



(a) Gradients assigned to integer coordinates.



(b) Closest four points and offset vectors.

Figure 3.7: Calculation of Perlin noise value in 2D, both sub-figures are adopted from [6].

For a more detailed explanation of Perlin noise, how it generalises to higher dimensions, and how exactly the gradients are chosen see the paper from Stefan Gustavson [6].

3.5 Jittered Grid

Jittering a regular grid is a form of stochastic sampling where noise is added to the sample locations. It can be used as an approximation of a Poisson disc distribution and guarantee that no two points are closer to each other than a set minimum distance. The basic approach to implement Poisson disc sampling is by generating pseudo-random points where every new point that is closer than a minimum distance to any of the previously generated points is discarded. New points are generated until the sampling region is full. This implementation is expensive. Faster implementations exist but get more complex, instead a Jittered grid can be used as a fast and straight-forward alternative with similar results [4].

Jittering of a regular grid was chosen as an object placement method in this thesis for both the islands in the sea and the objects on the islands themselves. For this purpose an extra step was added where noise values at the resulting sample positions are compared to a threshold and some of them are discarded. As the noise for both jittering and the thresholding was chosen Perlin noise because it is available by default in Unity. Computation of the positions for placing objects is done in the following three steps:

1. The placement positions are initialised on a square grid.
2. The positions are offset by Perlin noise.
3. Some positions are discarded by thresholding of Perlin noise.

In the first step, the positions are initialised on a square grid within a bounding box as illustrated in the figure 3.8a. The proximity between the placement positions can be adjusted by changing the spacing of the grid.

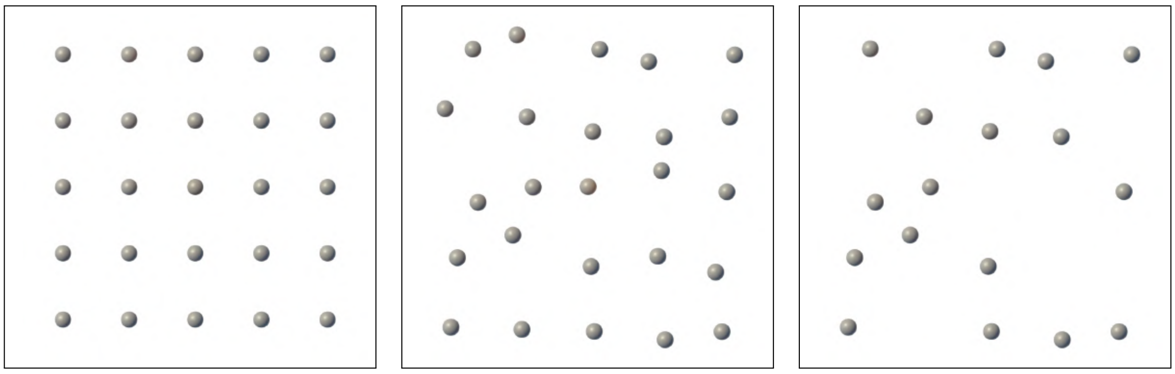

(a) One square grid. (b) Offset by noise. (c) After thresholding.

Figure 3.8: Three steps of the computation of object placement positions.

In the second step, each position is offset in the X and Z axis as seen in the figure 3.8b. The amount of offset in each axis is determined by the value of a Perlin noise sampled at the initial on-grid position. The noise value is between 0 and 1. After multiplying the noise value by a constant maximum offset o_{max} the range of the offset is from 0 to o_{max}. A separate Perlin noise with independent initialisation parameters is sampled to determine the offset in each axis. The positions that are offset outside of the bounding box are discarded.

Each remaining position is assigned a maximum object radius to prevent collisions with other objects as seen in the figure 3.9. Let s be the *grid spacing*, o_x be a position's X offset, and o_z be a position's Z offset. Then the maximum object radius r_{max} is defined as:

$$r_{max} = s/2 - max(|o_x|, |o_z|) \tag{3.1}$$

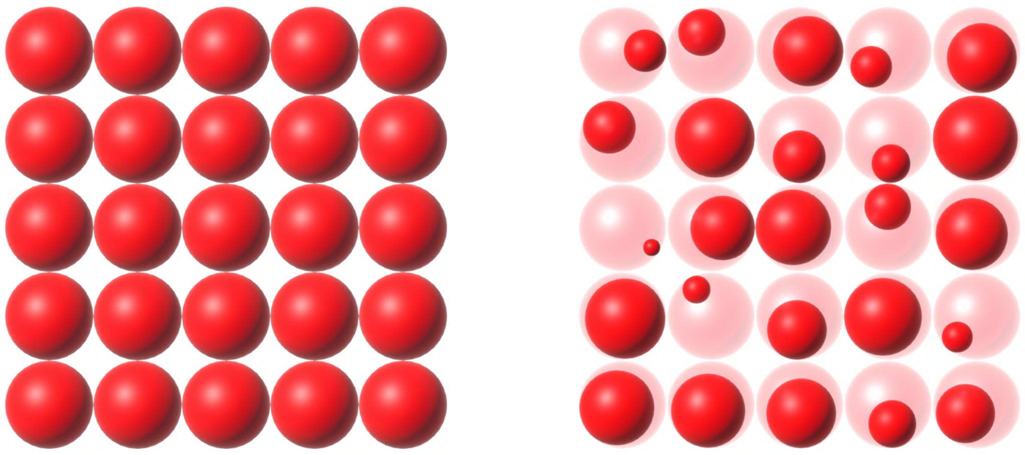


Figure 3.9: Maximum object radius before and after offsetting of the positions.

As the third step, some of the positions are removed by thresholding as seen in the figure 3.8c. This is achieved through sampling a third independent Perlin noise at the same on-grid coordinates of the initial positions as in the previous step. The value is compared to a threshold value. The threshold is a constant in the range of 0 to 1, same for all the positions. If the value of the noise is lower than the threshold, the position is discarded. The discarding of some positions achieves a sparse distribution with occasional clusters which resembles a natural distribution of objects.

3.6 Pseudo-random Number Generation

For procedural generation of repeatable random worlds a source of pseudo-random numbers is necessary. To serve this purpose there are often initialised several instances of a pseudo-random number generator (PRNG) for different tasks in the process of world generation. Using different numbers as seeds to initialize several instances of a PRNG is not enough to secure a pseudo-random (further called only random) relationship between the random numbers of same indexes generated by the different generators [7]. This problem is illustrated in the figure 3.10.

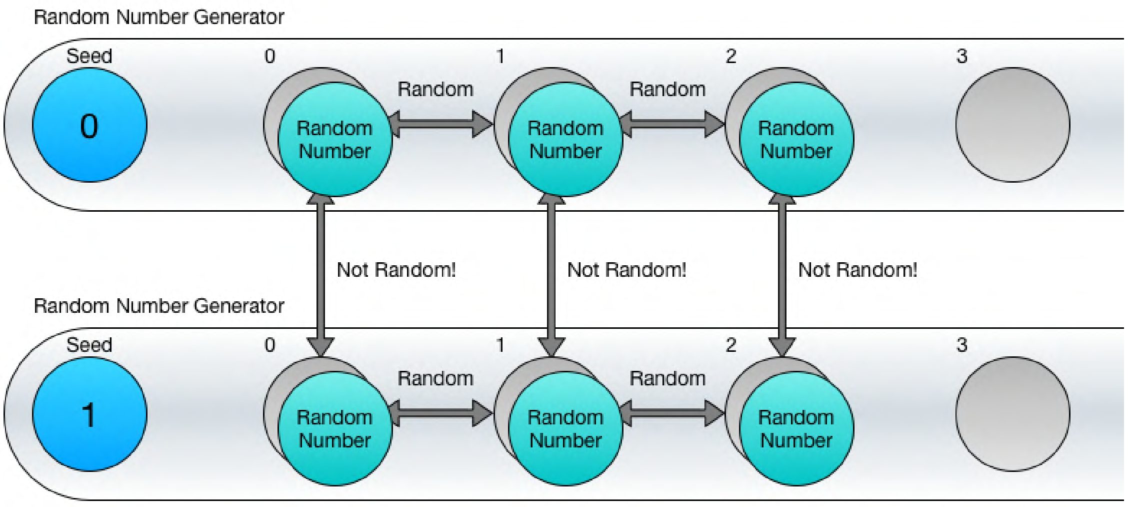


Figure 3.10: Relationship between random numbers, adopted from [7].

Within this thesis the placement positions of the islands in the sea are generated by a Jittered grid. Because of the fact that the positions originate from a grid and are further used as seeds for initialization of multiple instances of a PRNG in the island generation process, the problem of the relationship between the generated random numbers had to be addressed. Otherwise repetitive patterns might be observable on the generated islands.

Hash Function „xxHash“

To secure a random relationship between numbers from multiple random number generators a random hash function can be applied to the seed values before passing them to the PRNG for initialisation. The use of a random hash function solves not only the previously discussed problem, but also provides a way of combining several inputs into one integer seed. Otherwise the island coordinates would have to be combined in another way like summing the coordinates for every axis together which would result in many of the positions summing into the same seed.

As the specific random hash function used for these purposes was chosen the high-performing modern non-cryptographic hash function „xxHash“. Its pure C# open source implementation contained within a single file was chosen [12]. Combination of a random hash function with random number generators and how the generated numbers from multiple generators have a random relationship between each other is illustrated in the figure 3.11.

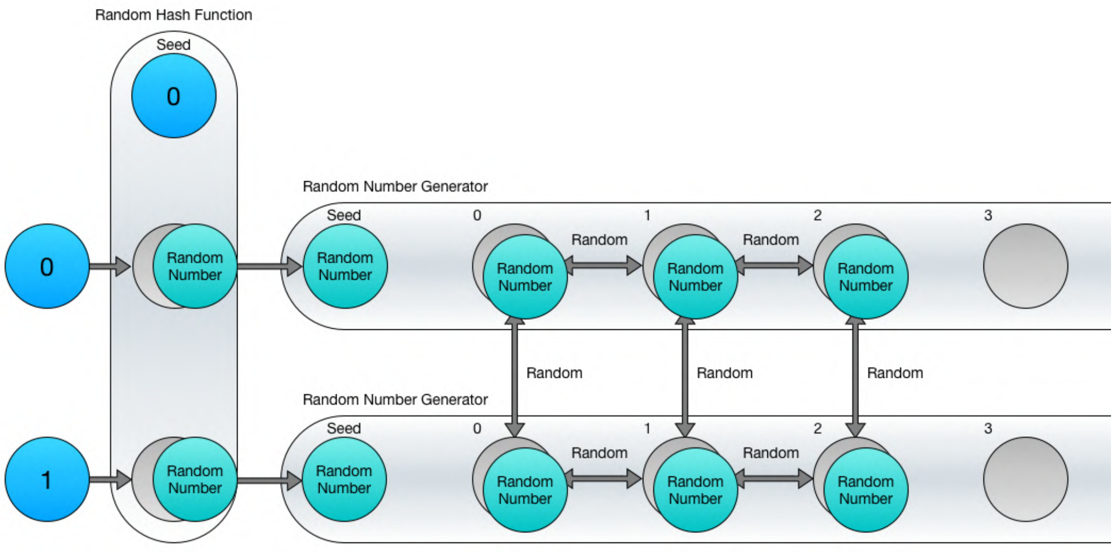


Figure 3.11: Hash function combined with random number generators, adopted from [7].

Chapter 4

Water Surface Animation

This chapter briefly introduces approaches to the animation of water surfaces and simulations of fluids in general. Both general methods and real-time suitable approaches are covered. In this thesis, a procedural approach to animating a sea surface is used. The effect is implemented through a fragment shader generated by Unity's *Shader Graph* tool.

4.1 Fluid Simulations

There are two main viewpoints on simulating the movement of a continuum like a fluid or deformable solid. They are called the Lagrangian and Eulerian viewpoint.

The Lagrangian viewpoint treats the continuum as a particle system where each point in the fluid or solid is modelled as a separate particle having its own position $\vec{x}$ and velocity $\vec{u}$. Solids are almost always simulated in a Lagrangian way, with a discrete set of particles usually connected in a mesh.

The Eulerian viewpoint is usually used for fluids and takes a different tactic. Instead of tracking the individual particles, at fixed points in space the change of quantities of the fluid in time is looked at. When the fluid is flowing past those points it contributes one sort of change to them.

Mathematically, simulations of viscous fluid substances are based on the incompressible Navier-Stokes partial differential equations which are usually written as:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{4.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{4.2}$$

The first differential equation 4.1 is called the *momentum equation* and it tells us how the fluid accelerates due to the forces acting on it. The second differential equation 4.2 is called the incompressibility condition. This section is based on the *Fluid Simulation SIGGRAPH 2007 Course Notes* [5] and a comprehensive explanation along with a break down of the previously mentioned equations can be found there.

Real-Time Simulation Requirements

In the context of real-time fluid simulations for interactive applications like games, a simulation algorithm needs to have the following characteristics:

- **Cheap to compute:** A computer game is expected to run smoothly at a stable frame rate around 60 frames per second. The total computation time of a single frame at such frame rate is around 16.6 ms out of which only a fraction can be used for the computation of water. Because of this a low computational complexity is one of the most important constrains for real-time simulations.
- **Low memory consumption:** In off-line simulations it is possible to simply buy as much memory as needed to simulate the desired effect. In contrast to that, most games and other applications running on single PC's or game consoles have limited amount of memory shared with other tasks. Therefore special attention has to be paid to the memory efficiency of the method too.
- **Stability:** Adaptive time stepping can be used in off-line simulations as a way to deal with arising stability problems in certain situations. On the other hand, games run at a fixed frame rate. Therefore the chosen method has to be stable for the given time step no matter what happens.
- **Plausibility:** Since it is not possible to reduce the computational and spatial complexity and at the same time increase the stability without any trad off with the quality of the result. Because of these reasons we require from a real-time simulation method to be visually plausible and wow effects are more important than a very accurate visually indistinguishable simulation from reality.

Real-Time Water

Because of the constraints listed above, solving the full 3D Navier-Stokes equations is not a practical way of simulating water in games. For simulating water in real-time the following three types of methods have become popular:

- **Procedural Water** Procedural methods animate the physical effect directly instead of simulating the cause of it. These methods are commonly used to procedurally animate water surfaces such as lakes, seas or oceans. An important advantage of these methods is their controllability which is a desirable property when creating games. A disadvantage of the procedural methods is the difficulty of getting the interactions of the water with immersed bodies and boundaries right. A procedural approach to animating the sea surface in this thesis has been chosen as it is fast and controllable.
- **Heightfield Approximations** For animating two dimensional water surfaces, it is needless to simulate the entire three dimensional body of water. For this case, only the surface can be represented as a two dimensional function or heightfield and animated using a 2D wave equation. Simplifying the simulation in this way makes it orders of magnitude faster. A limitation of these types of methods is the inability to simulate breaking waves because a function or a heightfield can only represent one height value at each location.
- **Particle Systems** As the other extreme, is the simulation of small amount of water like small puddles or splashes. In these cases a simplification method called a particle system is an interesting option. Height field approximations can be combined with particle-based fluids to create interesting effects.

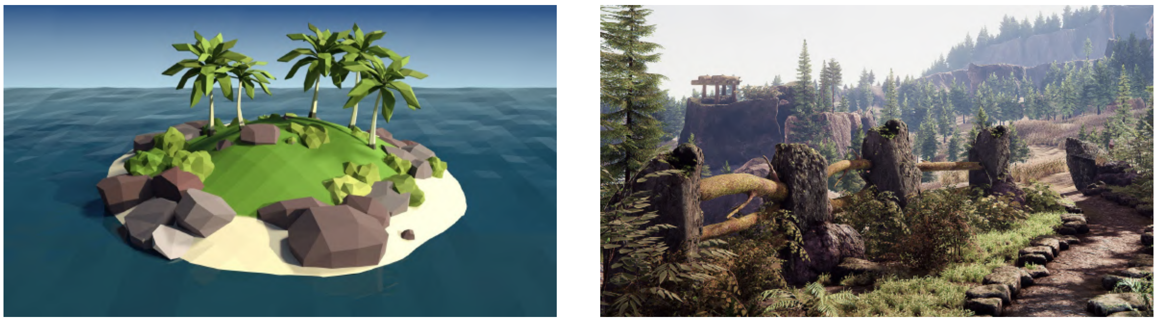
Chapter 5

Proposed Design

This chapter describes a design of a world generation tool and how it could be different from the existing solutions. After discussing the existing solutions and defining a detailed specification, designs of both a terrain generator and a water surface shader are presented.

5.1 Assessment of the Current State

After conducting research of the existing tools for procedural world generation available in the *Unity Asset Store*, two types of tools were examined. First, a generally oriented type of tools and second, tools focused on generating islands in the sea. Details about the research can be found in chapter 2. The generally oriented tools are very advanced and produce highly realistic landscapes as seen in the figure 5.1b. Instead of trying to compete with them in the domain of realistic landscape generation, it was decided to choose a different approach. The goal of this thesis is not to generate realistic terrain that is designed to be viewed from up close while walking on the ground as a character and thus needing to be very detailed and to include many game objects with high definition textures. The focus of this thesis is on generating a world that will be navigated by sea and viewed from a higher altitude, because the quality of the available tools targeted and optimised for this niche was significantly lower. In order to set a feasible goal for this thesis that could still be visually appealing, it was decided to stylise the tool in a cartoon like style as seen in the figure 5.1a, rather than trying to chase realism and ending up with an unsatisfying result. These decisions could give my tool a chance to become popular even in a saturated market by targeting sea oriented games with a cartoon like style.



(a) Cartoon style island visualisation [14] (b) Realistic landscape visualisation [17]

Figure 5.1: My target style on the left and realistic style on the right.

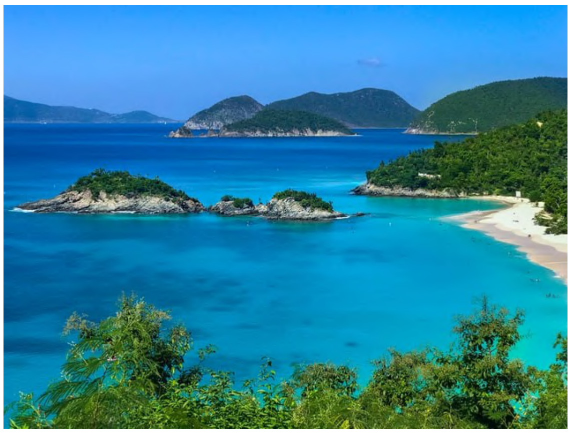
5.2 Detailed Specification

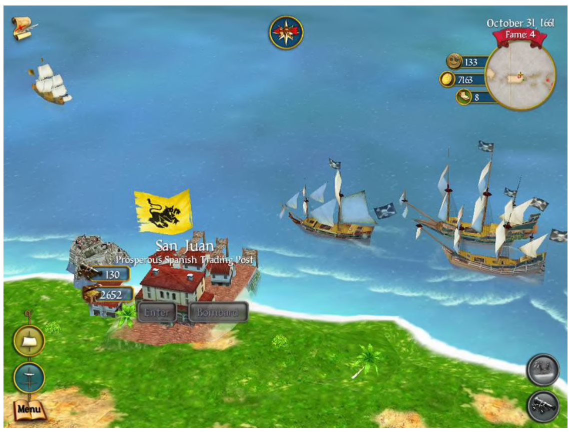
The aim of this thesis is to create a procedural terrain generator for generating a world with tropical islands scattered across a water surface, resembling islands that can be found in the Caribbean Sea, as seen on image 5.2a. This can be a basis for a top-down viewed game, where the player is sailing across the sea as a pirate fighting other ships and occasionally visiting a harbour on one of the islands. The result of my work should be a useful tool to help game developers create more sea oriented games such as the game *Sid Meier's Pirates!* shown in the image 5.2b, which was one of my favourite childhood games.

Non-technical Specification

The islands will be composed of beaches, rocks, palm trees, hills and human made structures like buildings and harbours. The sea's colour and transparency will change according to the water's depth. On the sea's surface an effect animating waves will be observable and sea foam will appear near the coast and around objects partially submerged in the sea. Objects or their parts under the sea surface will appear distorted to approximate refraction of light. Emphasis will be put on a pleasing visual appearance and not on a necessarily accurate and realistic representation of the real world. A list of the visual elements expected in the generated world is given below.

- Tropical islands with rocks and palm trees
- Houses and a harbour on some islands
- Depth dependent sea colour
- Sea with waves and sea foam
- Distortion effect imitating refraction of light


(a) Photo of islands in the Caribbean Sea [8]



(b) Screenshot from *Sid Meier's Pirates!* [16]

Figure 5.2: Images that served as inspiration to me.

Technical Specification

The terrain generator should be able to generate parts of a theoretically infinite world in real-time. Practically, the world size will be limited by the limitations of pseudorandom number generators, floating point data type precision and other technical restrictions. It will be possible to generate the part of the world around the player to cover the entire field of view of the camera. Every generated world part should be identically reproducible from a *seed*[1] dependent on the generator setting.

The objects automatically placed on top of the generated terrain such as trees, rocks and buildings will not be procedurally generated and their models will have to be provided to the generator. These object's positions will be calculated in a deterministic way depending on the terrain properties of the generated world part. Visualisation of the sea surface will be achieved through a custom *shader*.

The generator will be developed as a reusable component for the Unity game engine that will allow the option of publishing it on the *Unity Asset Store*. Below is a summary of the technical requirements that are expected from the final solution.

- Generate parts of an infinite world in real-time.
- World parts are identically reproducible when the same settings are used.
- Generate the parts of the world around the player to cover the camera's field of view.
- Deterministic placement of objects on top of the generated terrain.
- Custom *shader* for the sea surface visualisation.
- Implementation as a reusable component publishable on the *Unity Asset Store*.

[1] *Seed* will be the value used for the initialization of pseudorandom number generators.

5.3 Terrain Generator

This section presents the design of a terrain generator focused on generating tropical islands scattered across the sea. The general concept is presented in this chapter and the implementation details can be found in chapter 6.

World Partitioning

The world is separated into sea tiles and island areas. The sea tiles consist of a sea surface and a sea bed. At the same height as the sea bed of the sea tiles are placed independent island areas as seen in the figure 5.3.

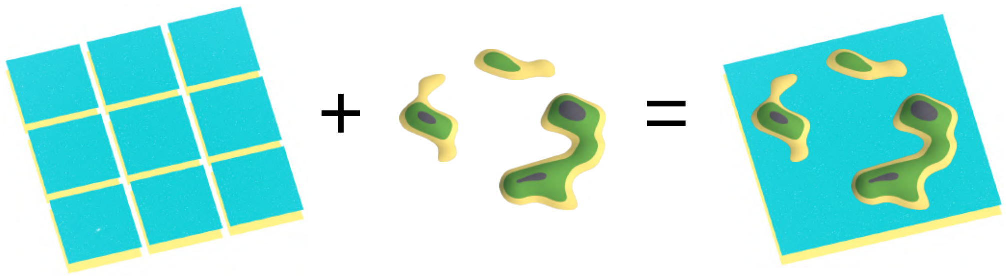


Figure 5.3: Combination of sea tiles and island areas.

Island areas can contain multiple islands or cliffs connected by a sea bed. The sea bed within the island area gets generated with full detail because some parts of it can be visible through shallow water.

On the other hand, the sea bed belonging to the sea tiles can be generated in a very simplistic way or can be completely omitted since it is not visible to the player through the deep water. That results in a more computationally efficient world generation optimised for this type of games.

As the main benefit of this partitioning of the world can be mentioned the possibility of having multiple different island types because the island areas are separate from each other and are not generated from a single continuous noise. Examples of the diversity of possible island types can be seen in the figure 5.4.

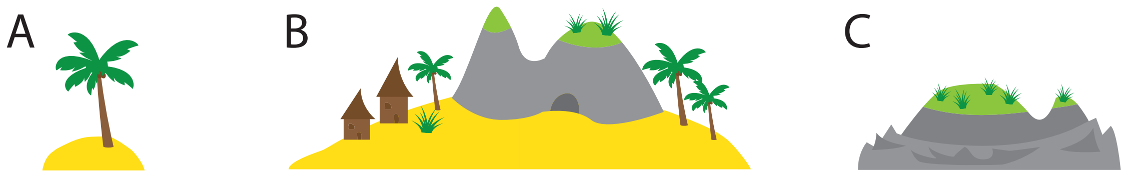


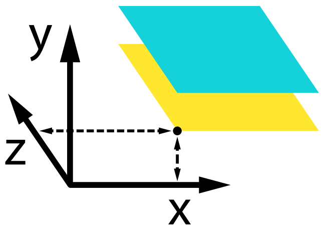
Figure 5.4: Example of the possible diversity of island types.

Another benefit is the absence of unwanted visual artefacts appearing on the seams of multiple joined terrain chunks and order of generation problems that have to be addressed in the traditional approach discussed in section 3.1.

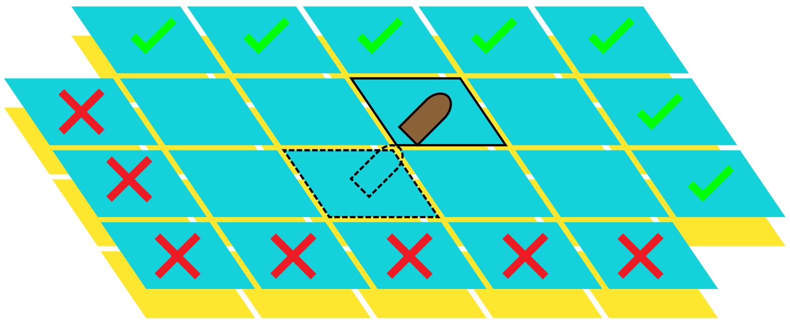
Generated Area

The generated area marks the part of the infinite world that should be currently generated. It is determined by the bounding box of the generated sea tiles. The sea tiles that should be generated are derived from the player's position. Within their bounding box are placed island areas generated on the positions of a jittered grid falling within it. Jittered grids and their advantages over other object placement methods are described in the section 3.5.

The sea tiles have a square shape and are placed on a square grid with a spacing equal to the length of a tile's side. A tile's position is determined as the distance from the world's origin in X, Y, and Z axis from the tile's bottom left corner of the sea bed plane as seen in the figure 5.5a.



(a) Tile's origin.



(b) Generated tiles around the player.

Figure 5.5

At first, the position of the tile on top of which the player is currently located is calculated. The current tile is generated along with other tiles around it. The number of other tiles next to the current one that get generated too is determined by a set distance in a number of tiles in the X and Z axis. Next, the tiles that are further away than the set distance from the current tile are destroyed as seen in the figure 5.5b. The whole process is repeated when the player leaves the current sea tile.

This approach is suitable for top-down viewed games where the camera is following the player's position with a constant offset. In this type of game a relatively small number of tiles can cover the whole area visible to the camera. If it is necessary for the camera to be able to observe the world independently of the player's position, another followed target could be simulated instead of the player.

Generating a Sea Tile

A sea tile consists out of two square planes above each other. One representing a region of a sea bed and second, on top of it, representing a part of a sea surface. The sea tile can be generated from two input parameters, the side length and the sea level height.

The position of the entire sea tile is measured from the bottom left corner of the sea bed plane. The default unity plane object with a yellow material has been used in the implementation but a simpler plane with a mesh constructed out of only two triangles is sufficient. This plane can also be completely omitted from generation, since it is not seen by the player through the layer of deep water and served mainly for demonstration purposes for the screenshots in this thesis.

A default Unity plane object is used for the representation of the sea surface. The sea surface plane has the same X and Z coordinates as the sea bed plane but is elevated to the input sea level height along the Y axis. Its animation effects are created by applying a texture to it, that is calculated by a shader described in the section 5.4. The mesh of the sea surface plane can also be constructed out of a higher amount of triangles than the sea bed plane, for the purpose of animation by manipulating the positions of the vertices. This has been tested, but negatively impacted the performance and was not evaluated as a suitable method for this type of game with a large water surface viewed from a distance. So the default Unity plane object without vertex animation was used after all.

Generating an Island Area

Each island area is generated from three input values. The first input is the position at which the island area should be generated. The second input is the maximum radius, defining the area around its position within which it has to be contained to not collide with other island areas. Lastly, the third input is a set of parameters defining the landscape and the quality of the texture and the polygonal mesh representing it.

Terrain Types

The terrain of the generated island area is composed out of several terrain types. Each type has a name, colour, starting height, probability of being dominant and noise parameters.

The starting height and the colour define what colour the terrain is going to be until it is overwritten by the next terrain type with a higher starting height. With a specified probability a terrain type can be dominant. A dominant terrain type's starting height is lowered to the starting height of the type below it. Therefore it dominates its bottom neighbouring type. A comparison of a terrain without and with a dominant terrain type can be seen in the figure 5.6.

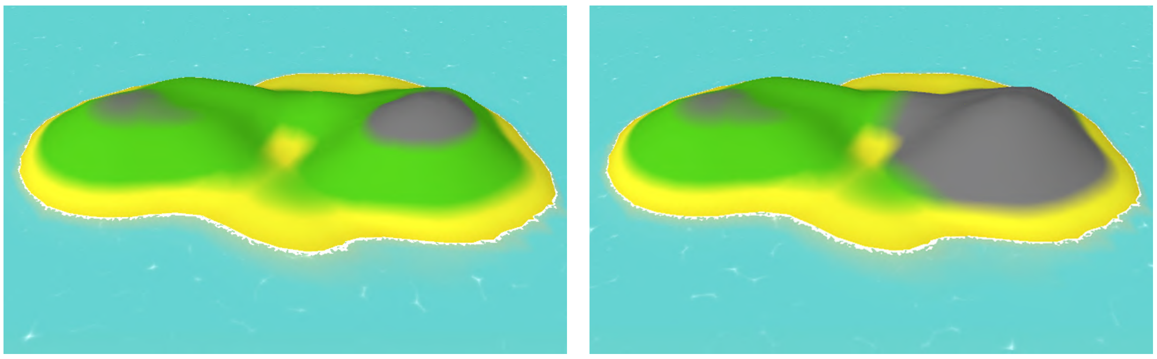

(a) Without a dominant terrain type. (b) With a dominant „rock“ terrain type.

Figure 5.6: Example of terrain type domination.

A terrain type also stores parameters of a noise used in the process of generating the heightmap defining the terrain's elevation. Perlin noise has been used within this thesis and it is appropriate for generating smoothly changing terrain such as beaches and grasslands. In the future, more types of noises can be added. Namely, for terrain types like rocks, Rigid fractal noise would be better suited because of its sudden changes in height and sharp ridges.

Island Types

Before the start of the generation process, each island area is pseudo-randomly assigned an island type. Island type groups together all parameters defining a set of similar island areas. It stores parameters such as name, probability of being assigned, pixels per unit defining the resolutions of all calculated textures, vertices per unit defining the number of vertices of the terrain mesh, terrain node parameters, a list of placed object parameters, maximum height of the island area, material used for rendering the terrain mesh (for example diffuse for tropical islands and specular for glaciers or wet rocks) and lastly a reference to a text file with all possible names that can be assigned to island areas of this island type.

Pseudo-random Number Generation

Generation of an island area is handled by a series of chained tasks. Some of these tasks require pseudo-random numbers. To make it possible to identically reproduce the same island area multiple times, the whole process has to be deterministic and the generation of all the pseudo-random values has to be based on an input seed value.

Each task in the generation process that needs random numbers initialises its own pseudo-random number generator (PRNG) with seed composed out of two input values. The first value is the world position of the island area and the second one is the name of the generation task. The position of the island area is converted to a string literal and concatenated to the name of the generation task. The resulting string literal is then converted to an array of bytes. This array is afterwards used as an input to the „xxHash“ algorithm that produces a hash used as the actual seed for the PRNG. The motivation to use a hashing algorithm is described in the section 3.6 and the „xxHash“ hashing algorithm is described in the section **??**.

It would be sufficient to initialise a single PRNG per island area that would be shared by all tasks, but it would introduce a dependence between them. This dependence would be caused by the single shared sequence of pseudo-random numbers generated by a single PRNG. Each task would get different pseudo-random numbers of that sequence depending on how many numbers have been required by the previous tasks and a modification of a task would influence all the other tasks executed after it.

Terrain Nodes

The terrain nodes are points generated within a radius smaller than the island area's radius. This maximum radius is marked by the red dashed circle in the figure 5.7a. Every node has an area of influence that is defined by a radius. The radius of a node's influence is calculated as the radius of the island area minus the distance of the terrain node from the centre of the island area as visualised in the figure 5.7b.

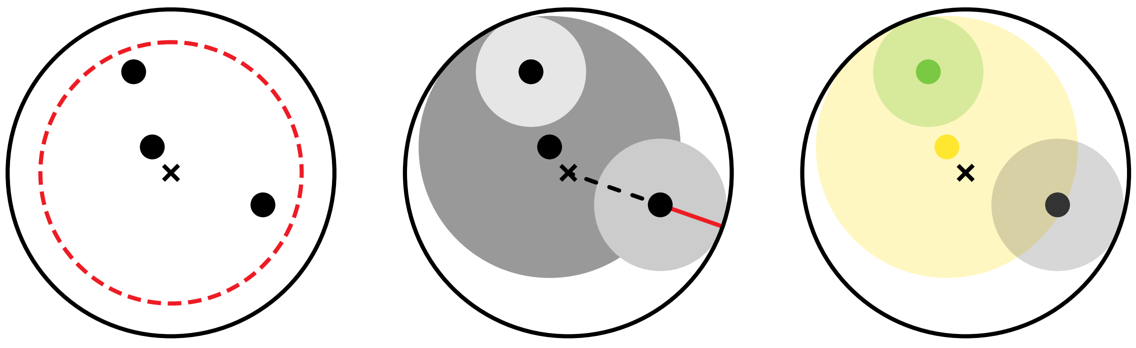

(a) Placement (b) Radius of influence (c) Terrain type assignment

Figure 5.7: Steps in the generation of terrain nodes.

Each node is also randomly assigned a terrain type as seen in the figure 5.7c. With the domination probability of the assigned terrain type it can become a dominant node. In the area of influence of a dominant node the terrain type that was assigned to it dominates its bottom neighbouring type as described in the section about terrain types.

Heightmap

The height map of an island area is a texture that defines the elevation of the terrain across the area. It is calculated in four steps with the help of the previously generated terrain nodes. Each pixel can have colours from black to white including different shades of grey. The brighter the colour the higher elevation of the terrain it signifies.

In the first step a separate height map is created for each of the previously generated terrain nodes. All of its pixels are set to black. Next, on top of each height map a radial gradient is drawn around the corresponding terrain node within its radius as seen in the figure 5.8. It is drawn by linearly interpolating from white to black depending on the distance from the node's centre. In order to calculate the colour of each pixel, the pixel's texture coordinates have to be converted to world coordinates from which the distance to the terrain node is calculated.

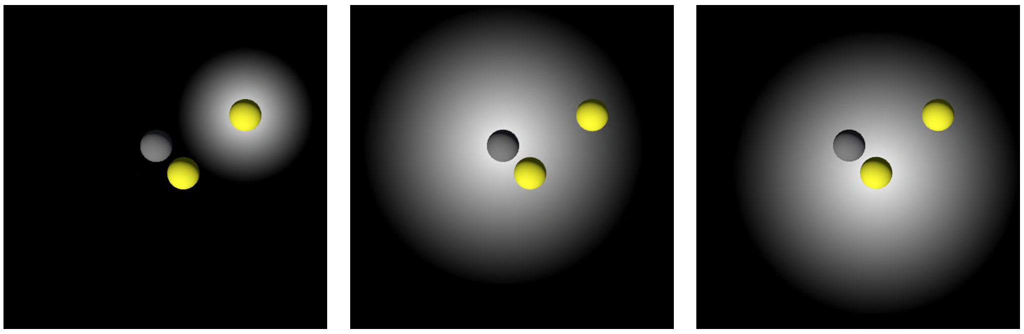


Figure 5.8: **step 1:** Generate gradient heightmaps.

In the second step, another set of heightmap textures are created. One for each terrain node again. This time, the colour of each pixel in the whole heightmap is determined by a noise value. The coordinates for sampling the noise values are determined as the normalised heightmap x and y pixel coordinates plus a random offset same for all the pixels within a heightmap. Let h_x be the heightmap x pixel coordinate, res_x be the number of pixels in the heightmap x axis, $U \in \langle 0, 1 \rangle$ be a random variable with a uniform distribution, and o_{max} be a constant maximum offset. Then the x coordinate of the noise n_x is defined as:

$$n_x = \frac{h_x}{res_x} + U * o_{max} \tag{5.1}$$

The n_y coordinate of the noise value is calculated respectively. A simpler calculation of the coordinates for sampling the noise has been tested too. The island area's coordinates were simply mapped to the noise coordinates. That approach produced visible mirroring of the heightmap around the world's origin. This unwanted effect was observed while testing with Perlin noise. The reason behind it was the Perlin noise symmetry across zero. Because of that, the offset was chosen to be a positive pseudo-random number generated from the island's position so it never crosses zero and the negative effect does not occur.

The noise coordinates are used to sample a noise defined by the noise parameters contained within the terrain type of the terrain node that the heightmap is calculated for. Examples of noise heightmaps calculated for each node can be seen in the figure 5.9.

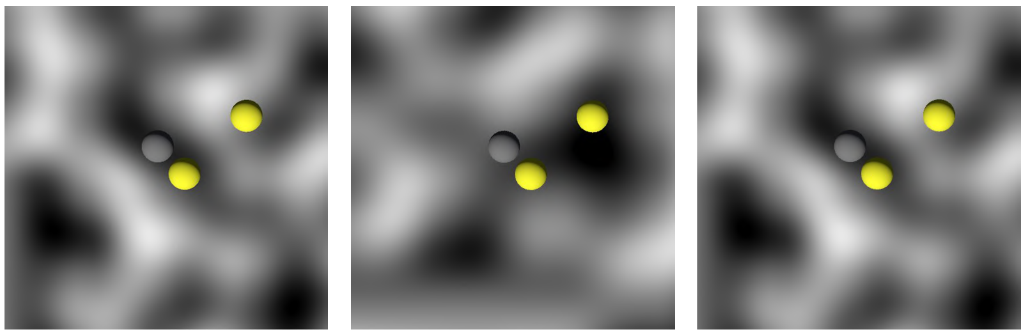


Figure 5.9: **step 2:** Generate noise heightmaps.

In the third step, each terrain node's radial gradient heightmap and noise heightmap are multiplied. This results in a single heightmap for each terrain node where the noise is gradually faded to black proportionally to the radial gradient, seen in the figure 5.10.

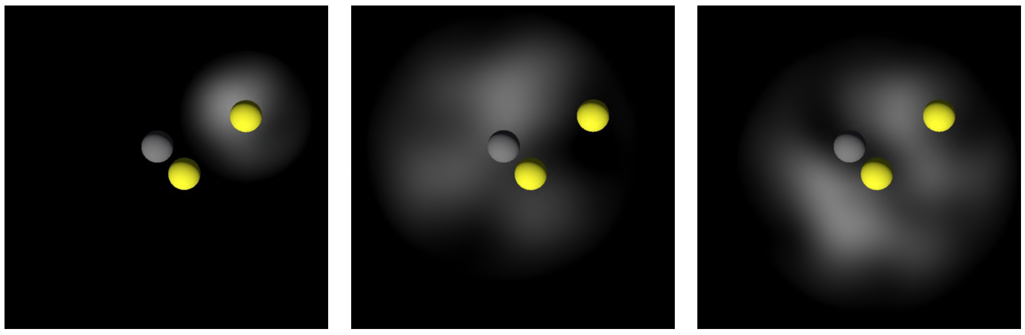


Figure 5.10: **step 3:** Multiply node's heightmaps.

In the fourth step, the results of the previous multiplication are added together into a single heightmap as seen in the figure 5.11. The added fading noises result in a final heightmap representing height values across the island area.

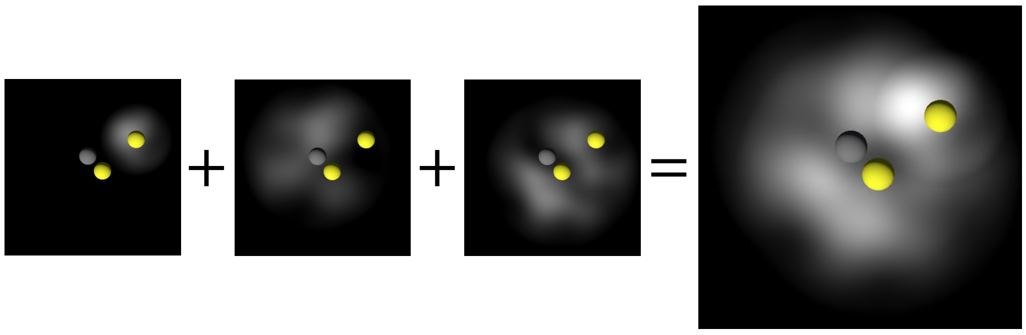


Figure 5.11: **step 4:** Add multiplication results together.

Terrain Texture

The texture applied to the terrain mesh is directly visible by the player and areas of different colours represent different terrain types such as sand, grass, rock etc. The resolution of the terrain texture matches the resolution of the heightmap and is defined by the parameters of the island type.

The creation of the terrain texture is achieved through a combination of several mechanics. These mechanics calculate a terrain blend for each pixel in the texture. Each blend has a list of terrain fractions specifying terrain types and their quantities in the blend. Afterwards the colour of each pixel in the terrain texture is calculated as a weighted average of colours belonging to the terrain fractions of the pixel's terrain blend.

The basic mechanic for determining a terrain blend for a certain pixel is done by height. This mechanic does not solve the blending of multiple terrain types, but assigns a single terrain type to the terrain blend instead. The assignment of a terrain type is done through a comparison of the value of the corresponding pixel in the previously generated heightmap to all the terrain types defined for this island area's island type. The terrain type with the highest starting height that is still below the pixel's height is chosen. This approach of assigning a single terrain type by height to each pixel results in unnaturally looking terrain with stripes and sharp transitions as observable in the figure 5.12a.

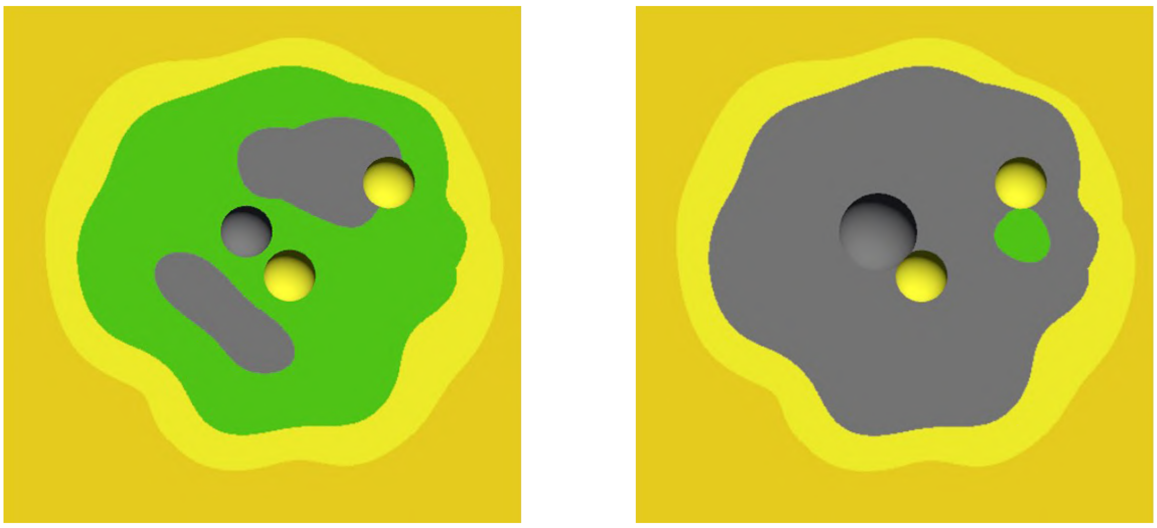

(a) Only height based mechanic. (b) Added dominant node.

Figure 5.12: Height based mechanic.

The next mechanic that builds on top of the first one is terrain node dominance. During the process of generating terrain nodes they have a probability of becoming dominant. This probability is equal to the domination probability of the terrain type assigned to them. Dominant terrain nodes force their terrain type to dominate its neighbouring type underneath it as previously described in the „Terrain types“ subsection. This domination effect is limited to the node's area of influence that is defined by the non-black pixels in the node's individual heightmap calculated by the multiplication of its radial gradient and its noise heightmap described in the „Heightmap“ subsection. The texture generated by the first mechanic shown in the figure 5.12a changes after the addition of a dominant node to the texture seen in the figure 5.12b.

The last implemented mechanic of the terrain texture generation is the blending of terrain types. In the island type's parameters it is possible to specify a blending height. This height defines a region between two neighbouring terrain types where the terrain types are linearly interpolated from one to the other. Directly on top of the border between the two types, each has a 50% influence on the colour of the pixel. A special case is blending within the radius of one or more dominant nodes. When the height of the pixel in the dominant terrain node's heightmap is lower or equal to the blending height, the terrain blend assigned to that pixel is a mix between terrain blends of dominated terrain types of dominant nodes in range. These terrain blends are mixed with weights proportional to the height in the heightmap of their respective dominant terrain node. If they do not add up to 100%, the remaining terrain blend is the one of the default terrain types without domination. On the other hand if the height of the pixel in the node's heightmap is above the blending height then the dominant node's terrain blend determines the pixel's colour entirely. This blending ensures smooth transitions between different terrain types and also between areas of influence of dominant terrain nodes as seen in the figure 5.13.

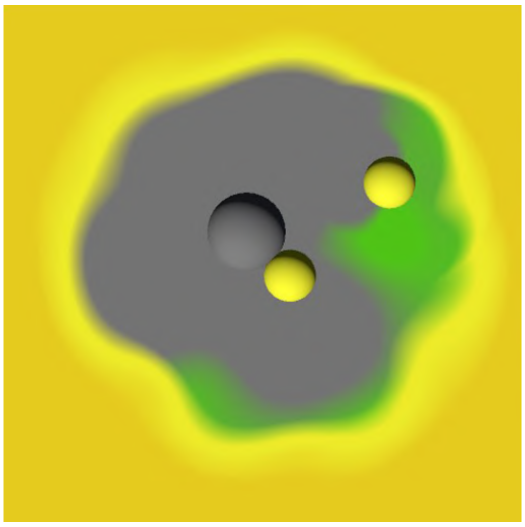


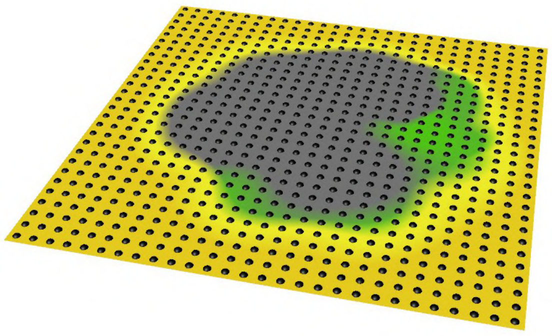
Figure 5.13: Added blending.

An extra technique has been designed, but it has not been implemented or tested within the scope of this thesis. It serves for enforcing a specific terrain type such as rock on steep surfaces for a more natural appearance. First, all steep surfaces have to be detected. This can be achieved by applying an edge detection algorithm such as a Sobel filter to the heightmap of the terrain area and saving the result into a separate steepness texture. Afterwards the pixels on the same positions as the steep pixels in the steepness texture would be forced to have a specific colour such as a rock terrain type colour. In the case that the transitions between the steep terrain type and other terrain types would be too sharp, the whole terrain texture could be slightly blurred by a filter such as a Gaussian blur.

Mesh

Input values for constructing the polygonal mesh of the terrain area are the dimensions of the island area, the heightmap, the terrain texture, and the vertices per unit meaning how detailed the mesh should be.

At first, the vertices are evenly spaced across the island area at zero height as seen in the figure 5.14a. Next, each vertex is elevated to the value of the corresponding pixel in the heightmap multiplied by the maximum terrain height as visible in the figure 5.14b.


(a) Evenly spaced vertices at zero height.

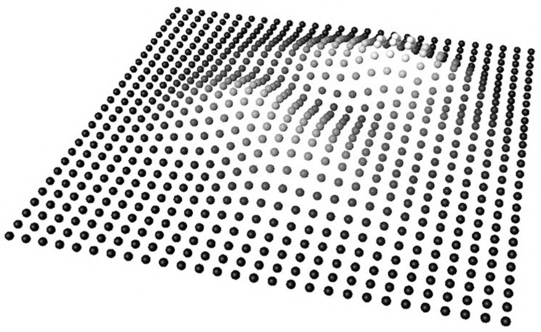
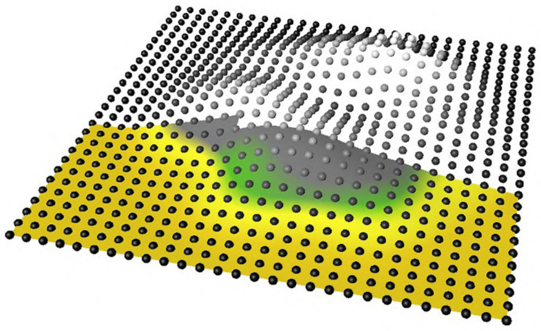

(b) Vertices translated to target height.

Figure 5.14: Vertices positions calculation visualised.

Afterwards, every set of four neighbouring vertices from neighbouring rows are connected to form a pair of triangles in a clockwise direction as seen in the figure 5.15a.

Lastly, the terrain texture is applied to the constructed terrain mesh. The resulting terrain mesh can be seen in the figure 5.15b.


(a) Connecting vertices to form triangles.

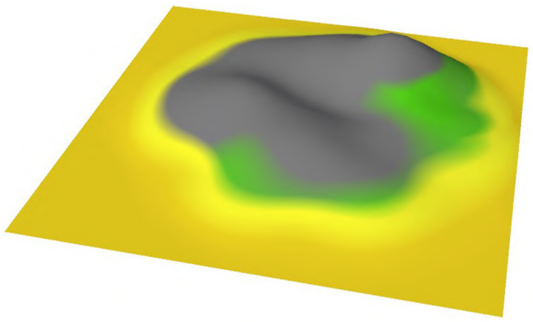

(b) Final island area mesh.

Figure 5.15: Connecting vertices into triangles and the final mesh.

Object Placement

Every island area can be given a list of object placement parameters, one parameters item for each object type that should be placed on the island area. Object placement parameters define the minimum and maximum height at which it can be placed, the minimum required fraction of a terrain type that has to be present at the object's position (e.g. 50% grass), and parameters defining the jittered grid on which the objects get placed. For each placed object type a separate jittered grid is used for placement. The scale of the placed object corresponds to the amount of offset of its jittered position. The usage of multiple independent jittered grids results in the possibility of colliding objects.

Object collisions can be avoided if every placed object type requires a different terrain type. If multiple object types need to be placed on the same terrain type without collision, another approach has to be devised. A possible approach could be a single jittered grid where different positions would be occupied by a specific object type depending on the maximum non-colliding radius of the object in that position.

The positions on the jittered grid determine the X and Z coordinates of the placed objects and their height in the Y axis is retrieved from the corresponding pixel in the heightmap of the island area multiplied by the maximum height of the terrain. This approach produced occasional visual artefacts in the form of „floating" objects. Some of the placed objects were occasionally placed above the terrain mesh. The cause of this phenomenon was found out to be a mismatch in the pixels of the heightmap and the number of vertices representing the mesh. Specifically in the cases where the number of vertices was lower than the number of pixels of the heightmap. The solution to this problem was changing the heights of all the objects to the average of the heights of the closest four vertices of the mesh from the original object position. Examples of island areas with several different types of placed objects are shown in the figure 5.16.

An extra technique for placing bigger objects like human made structures has been designed, but it has not been implemented or tested within the scope of this thesis. Such objects often need the underlying terrain to be modified in order to enable the possibility of placing the object there. For example a house needs a flat area where it can be built. Before placing such an object on an allowed terrain type, it would be necessary to modify the heightmap and the texture of the island area in a way so within the radius of the object the height and the terrain type would be constant. Afterwards the terrain mesh would have to be generated again or this step could be done before the mesh generation.

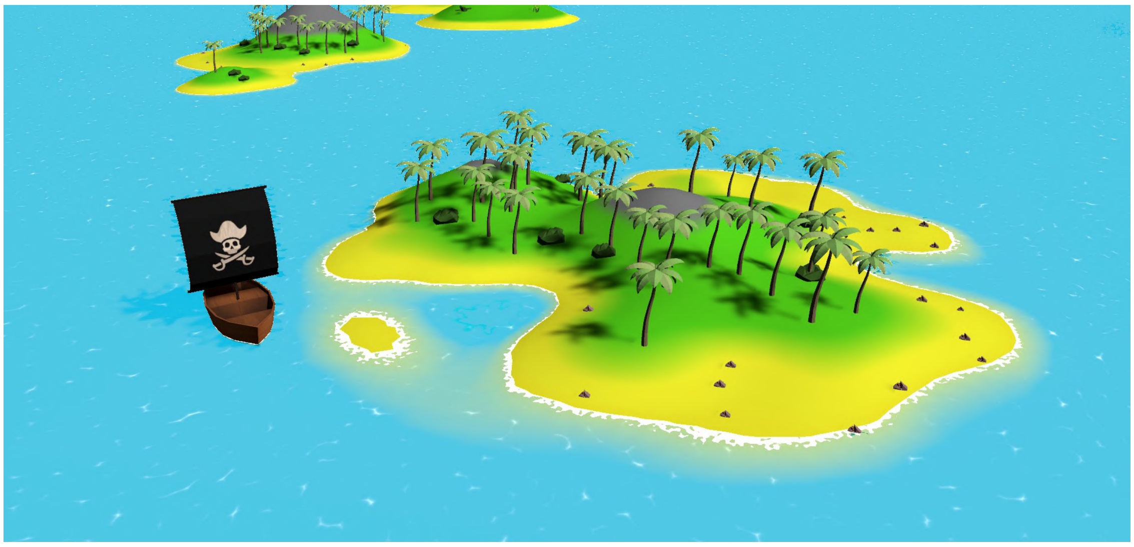


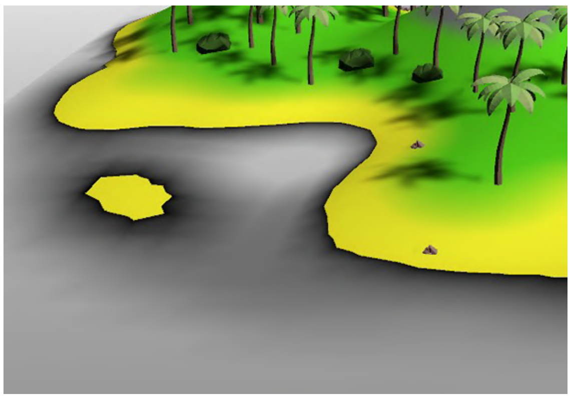
Figure 5.16: Island areas with several placed object types.

5.4 Water surface shader

The animation of the sea surface between the islands is achieved through a fragment shader. In this section it is broken down into individual parts that create the final effect.

Depth Based Colour

The first partial effect is changing the water's colour based on the water's depth. This is done by defining a shallow and a deep colour, each having its own transparency value. The colour of the water at a specific pixel is the result of a linear interpolation between the shallow and the deep colour based on the value of the camera's depth buffer divided by a maximum depth and clamped to values between 0 and 1. The transparency value of the resulting colour is used for another interpolation between the previously calculated colour and the scene colour to make objects below the water surface visible in shallow areas. In the figure 5.17a is visualised the depth buffer used for interpolation and in the figure 5.17b is shown a demonstration of the resulting effect.


(a) Depth buffer visualisation.

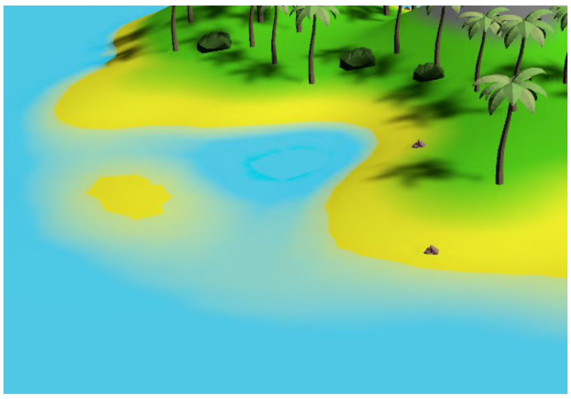

(b) Depth based water colour effect.

Figure 5.17: Visualisation of the depth buffer and the interpolated colours.

Sea Foam

The next effect added to the shader is a simplistic cartoon visualisation of sea foam in shallow areas, for example near the coast of an island. This effect also uses the depth buffer. The buffer's value is multiplied and compared to a gradient noise that is being offset over time to make the foam keep changing its shape. Only the noise values that are greater or equal to the buffer's multiplied value are drawn on the water surface plane. The foam is visualised with a white colour on the water surface. The isolated effect is seen in the figure 5.18a and the result after adding it to the previous effect is shown in the figure 5.18b. Both figures are located on the next page.

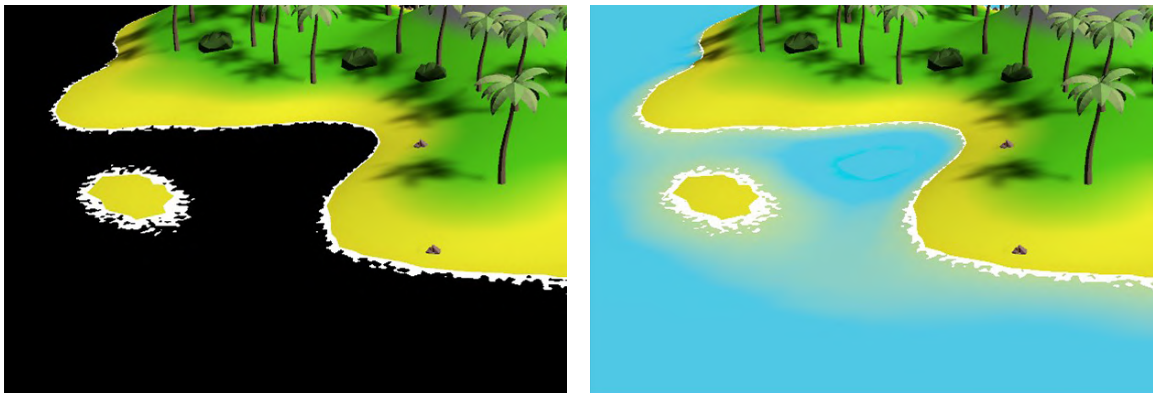

(a) Isolated sea foam effect. (b) Added to previous effect.

Figure 5.18: Visualisation of the sea foam both isolated and mixed.

Waves

A simplistic visualisation of waves is based on a Voronoi diagram. Through several steps it is made to mimic the foam of breaking waves. At first a basic Voronoi diagram is generated and its points are pseudo-randomly offset over time to resemble the movement of waves. The UV coordinates for mapping the noise onto the plane object are deformed by a radial shear to make the noise movement less uniform. The value of the noise is afterwards raised to the power of a user specified integer, that makes only the brightest values of the noise stay visible. The result is multiplied by a value between 0 and 1 to specify the transparency of the waves. As the final step the calculated texture is added on top of the previous effects as seen in the figure 5.19b. An isolated version of the effect is seen in the figure 5.19a.

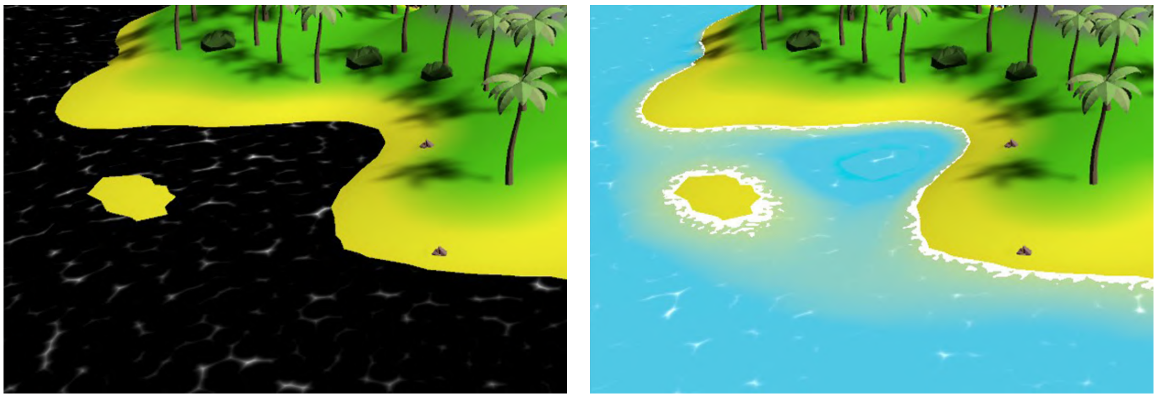
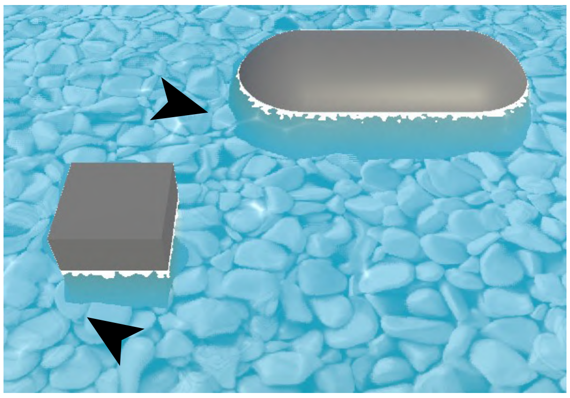

(a) Isolated waves effect. (b) Added to previous effects.

Figure 5.19: Visualisation of the waves both isolated and mixed.

Distortion

The effect of objects appearing distorted under a water surface related to light passing through mediums with different optical densities such as water and air is very roughly approximated through a distortion of UV coordinates. The UV coordinates of the screen colour used in the depth based water colour effect is distorted by gradient noise. Before adding the gradient noise to a screen position an offset proportional to the elapsed time from the beginning of the game is added to it. Because the effect is not clearly visible in the scene previously used for demonstration the effect is shown in scene with a higher water transparency in the figure 5.20a. The effect is also observable on the shadow of the boat in a previous scene when viewed from a different angle, demonstrated in the figure 5.20b.


(a) Distortion of the submerged parts of objects.

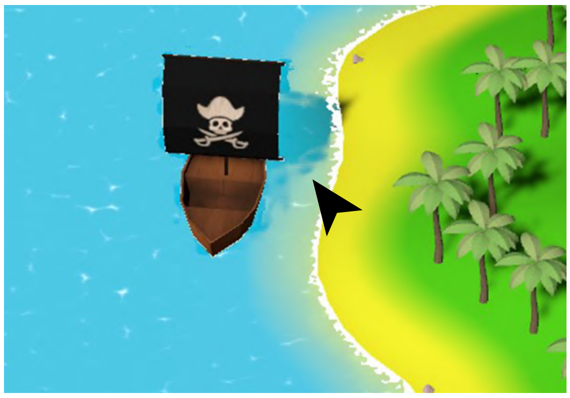

(b) Distortion of the boat's shadow.

Figure 5.20: Visualisation of the distortion effect.

Chapter 6

Implementation

The two main parts of this project are the terrain generator and the water surface shader. Since the terrain generator is more complex, it is further divided into two components. The first, being the sea tile grid and the second, being the island generator.

Both parts of the project are implemented in the game engine Unity, version 2021.1.1f1 and with the use of the IDE Visual Studio 2017 and its productivity plugin ReSharper[1].

The terrain generation part is developed in the C# programming language that is considered the main language for Unity. The water surface shader is created in the visual shader creation tool called the Unity Shader Graph[2].

An open source implementation of the „xxHash“ algorithm [12] and several 3D models [21] [9] [11] [10] created by others were used, they can be found in the folder „3rd Party“.

6.1 Architecture

Programming in C# for Unity is quite specific. The standard object oriented programming (OOP) design has to fit within the context of components because Unity is using a component based architecture. Two of the basic building blocks of a Unity project are game objects and components. Different components can be attached to game objects to form a complex behaviour. Creation of a custom component is done through creating a C# script containing a class derived from Unity's `MonoBehaviour` class. Such a script represents a component and can be attached to different game objects in the Unity Editor.

To maximise the re-usability and maintainability of the code base the following efforts have been made. The component scripts have been kept reasonably short and as much of the general functionality has been extracted to other classes that are not necessarily Unity specific. Components can easily become less reusable if they rely on instances of other components. A way to avoid tight coupling is to pass data between components through scriptable objects as described in the Unity's article[3]. This way of passing data has been used between the „Sea tile grid“ and the „Island generator“ components. The sea tile grid saves its bounding box to a scriptable object that is assigned as the area for generating islands to the island generator instead of referencing the tile grid directly.

[1]ReSharper plugin: https://www.jetbrains.com/resharper/

[2]Unity's visual shader creation tool: https://unity.com/shader-graph

[3]The article on architecture: https://unity.com/how-to/how-architect-code-your-project-scales

6.2 Sea Tile Grid

The sea tile grid is composed out of tiles containing parts of the sea surface and the sea bed as briefly described in the section 5.3. The sea tiles have a square shape and are placed on a square grid tightly next to each other to create the appearance of a continuous plane. The tile that the player is currently on top of, will be referred to as the *player's tile*. A specified number of tiles in the X and Z axis from the *player's tile* are instantiated and kept in the scene, while being in range. The implementation of this functionality can be broken down into three steps. First step, finding the position of the player's tile by dividing the player's position by the tile size and adding an extra tile length in the case of negative coordinates. Second step, finding the positions of the rest of the tiles in range by adding tile lengths to the position of the *player's tile*. Third step, instantiating sea tiles on the new positions and destroying the sea tiles out of range. The `SeaTileGrid` component uses the generic class `TileGrid` to instantiate sea tiles by the use of the `SeaTile` class, but can place any kind of tiles that are derived from the base class `Tile`.

6.3 Island Generator

Island generation forms the biggest part of this thesis and is split into a list of dividable tasks. The class diagram of all the tasks is attached in the appendix A. The island generator receives many input parameters for configuration, they can be found in the appendix B. In this section will be explained the custom built task management system and the blending of terrain types in the texture generator since they posed an interesting implementation challenge.

Task Management

Generation of an island is a complex task. During the testing of the first prototypes it was observed that generating even a simple island was taking over a second and unpleasantly freezing the game. Also it was assumed that it would be beneficial for developers to have the option to easily add, remove or modify steps of the generation process.

Since it was desirable to split the generation process into multiple tasks and some of these tasks were still computationally intensive, the classes `TaskList` and `DividableTask` were designed and implemented.

Dividable Task

The class `DividableTask` provides the mechanism for spreading the execution of a task over multiple frames in the game loop which makes it possible to sustain a playable frame rate and avoid freezing of the game. It is desired to use outputs of some tasks as inputs to subsequent tasks, but the output values are not calculated yet at the time of defining the tasks. This problem has been solved by passing a delegate as the input to a task that can reference to a `GetResult()` method of a previous task.

When a new task class is created it has to inherit from the abstract class `DividableTask` and implement its three abstract methods. The `GetInputFromPreviousTask()` method is the first one and in its definition the outputs of the previous tasks can be retrieved by calling one or more methods referenced by input delegates. Next, the `SetSteps()` method has to be implemented where the total and remaining steps of the task have to be set either based on the inputs to the task or by an internally computed value. The last abstract method that has to be implemented by a task is the `ExecuteStep()` method. This method defines how a single step of a task is executed. If it is desired to retrieve a task's output a method such as `GetResult()` can be additionally implemented.

To demonstrate the relationship between the specific tasks and the base class on a practical example, a simplified class diagram of the `DividableTask`, `AddTextures` and the `MultiplyTextureLists` tasks is presented in the figure 6.1.

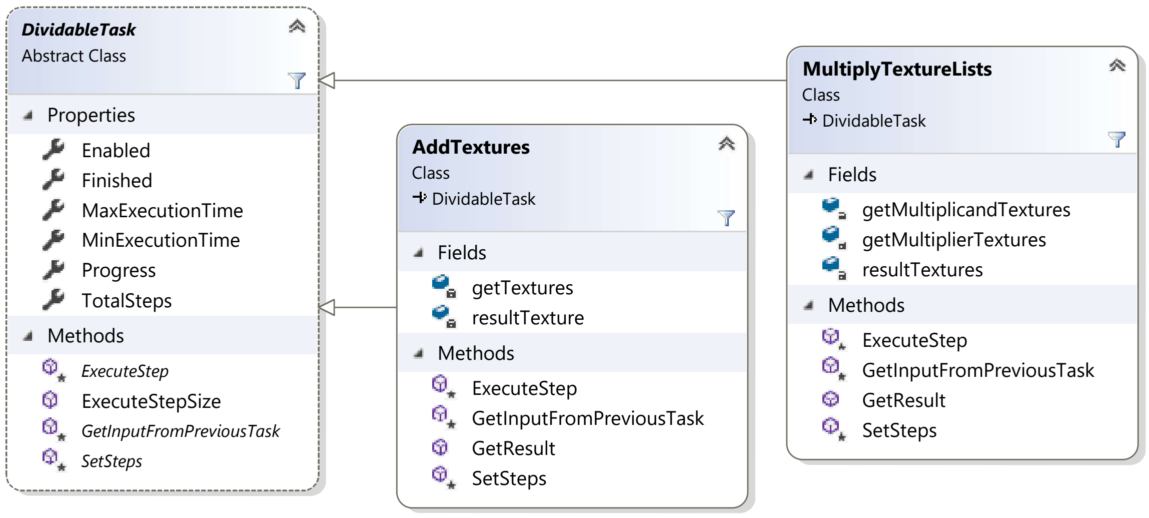


Figure 6.1: Simplified class diagram of the divided task and its descendants.

The task `MultiplyTextureLists` is executed first. It takes two lists of textures and multiplies the pixel values of the textures with corresponding indexes and saves the result to a single list of textures. The second task `AddTextures` has to retrieve the list of textures to be added together by calling the delegate `getTextures()` referencing the `GetResult()` method of the previous task.

Task List

The `TaskList` class makes it possible to chain multiple predefined tasks together, manages their execution and monitors the overall progress. Some tasks in the task list are used only for visualisation purposes and outside of a demonstration mode they are not necessary. As an example can be mentioned the `ShowTextures` task, used for showing internally computed textures during the process of island generation. Such tasks can be easily disabled just by setting their `Enabled` property to false without the need to modify the task list. These visualisation tasks shall also be able to display the visualisation for a long enough time, so they can be examined by the user. This is achieved by setting a minimum execution time to these types of tasks. If they finish earlier, they simply wait by executing steps of zero size before finishing.

A key mechanism deciding on how many steps will be executed when the `ExecuteStepSize()` method of a task list is called hides within the individual tasks. Every task can be assigned a maximum execution time for example 10 milliseconds and it keeps executing steps until this time is reached or the task is completed. That dynamically adjusts the step size according to the hardware it is running on.

Parallel Execution of Tasks

Even though the tasks are currently executed sequentially, they can be extended in the future to support asynchronous execution. The tasks were not implemented to support parallel execution by default because of the specificity of Unity and the lack of time to learn about Unity's job system[4]. Because Unity is not thread safe it has disabled the option to call its API outside of the main thread. This fact forces tasks that deal with instantiating new objects or translating their positions to be executed on the main thread. Luckily, some tasks can be split into the computation heavy part and the part using the Unity API. As an example can be mentioned the most computation heavy tasks for generating textures. These can be upgraded to compute the texture data in parallel on other threads and the textures can be applied to game objects afterwards on the main thread.

Different options for making tasks run in parallel were researched. The language C# by itself supports either creating threads or asynchronous tasks. Threads can be created through the `Thread` class where the developer has to manage the communication between threads and deal with race conditions. On the other hand, from the version of .NET Framework 4.0 it is possible to use the `Task` class that makes it easier and safer to write asynchronous and parallel code.

Even though the above stated options are available when programming for Unity, to achieve the best performance Unity recommends using their job system in combination with their *Burst*[5] compiler. Another option for an even higher performance boost of tasks like the ones generating textures is the option to use compute shaders. Although it should be considered that they are suitable only for some tasks and narrow down the supported target devices as some mobile devices do not support compute shaders. In the future the use of Unity's job system and the *Burst* compiler to improve the performance of the developed island generator seem as an obvious choice.

Texture Generator

The texture generation functionality is contained in the `TerrainTextureGenerator` class. The name of the method that gets called from the task responsible for the texture generation is `GetPixelTerrainBlend()`. It calculates the blend of terrain types for a given pixel defined by a height from a height map and an array index of a flattened 2D pixel array.

A pixel's terrain blend is computed by a combination of several mechanics. Firstly, an ordered list of terrain types ordered by starting height is created out of the terrain types in the island type's terrain node parameters. Next, if the pixel is within the range of a dominant node a modified (dominated) version of the terrain types list is created for each dominant node in range. The difference between a regular ordered list and a dominated one is that the terrain types directly below a dominant type are replaced by the dominant type above that adopts the below type's starting height. Based on the height of the pixel an instance of the `TerrainBlend` class is created for each of the lists. A terrain blend is created by adding instances of the `TerrainTypeFraction` class to it which represent just a terrain type with a specified amount between 0 and 1. After a `TerrainBlend` is calculated for each of the ordered lists, all of the blends have to be mixed into a single terrain blend. That is secured by the `TerrainBlendList` class which holds a list of terrain blends and proportionally to each blend's amount adds its terrain fractions into a single terrain blend.

[4]Unity's C# Job System: https://unity.com/dots/packages#c-job-system
[5]Unity's *Burst* compiler: https://unity.com/dots/packages#burst-compiler

Therefore the `TerrainBlendList` performs the sum of weighted terrain fractions. In the end, the calculated terrain blend for a certain pixel is used to determine the pixel's colour. The terrain blends for all pixels are saved because the task for placing objects uses the earlier computed terrain blends for deciding if a pixel has a high enough terrain fraction for placing an object on its corresponding position. For a more clear understanding of the terrain blending it is illustrated on a simple example in the figure 6.2.

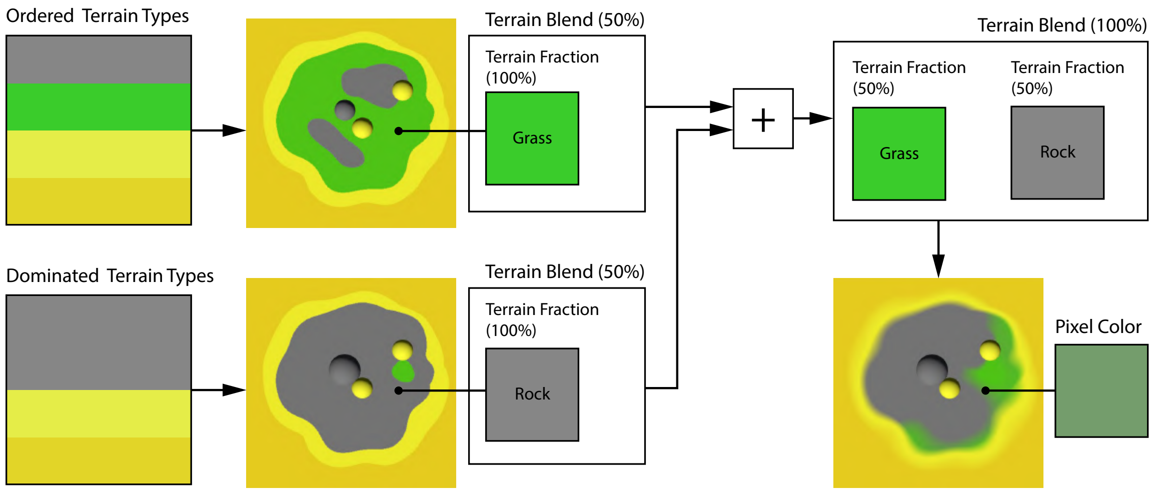


Figure 6.2: Blending of terrain types in the texture generator.

6.4 Water Surface Shader

Unity's *Shader Graph*[6] tool enables quick shader creation and live preview of the results. This tool requires a Unity project that is configured to use the *Universal Render Pipeline*. Because *Shader Graph* is relatively new some of its bugs were encountered that made it impossible to divide the water surface shader into sub-graphs. Instead, the individual effects in the graph were divided into groups. The concept behind every effect is explained in the section 5.4. The full shader graph is attached in the appendix C and a simplified diagram showing how the effects are combined together is presented in the figure 6.3.

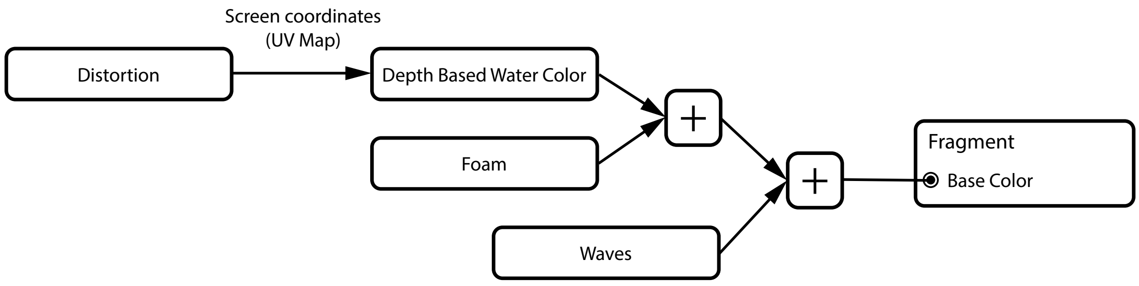


Figure 6.3: Simplified shader graph.

[6]Unity's *Shader Graph*: https://unity.com/shader-graph

6.5 Showcase Application and Testing

An interactive application showcasing both the island generator and the water surface shader has been built for the Windows desktop platform meeting the requirements listed in table 6.1 and a web browser variant based on WebGL supported by the devices complying to the requirements in table 6.2. The builds can be found on the attached medium to this thesis and the WebGL variant is also available online[7].

Operating system	Windows
Operating system version	Windows 7 (SP1+) and Windows 10
CPU	x86, x64 architecture with SSE2 instruction set support.
Graphics API	DX10, DX11, DX12 capable.
Additional requirements	Hardware vendor officially supported drivers.

Table 6.1: Desktop platform requirements of the showcase application [22].

Operating system running browsers	Windows, macOS, and Linux
Hardware	Workstation and laptop form factors.
Additional requirements	Versions of Chrome, Firefox or Safari that are: • WebGL 1.0 or 2.0 capable • HTML 5 standards compliant • 64-bit • WASM capable

Table 6.2: WebGL platform requirements of the showcase application [22].

Performance Testing

Both the desktop and the web versions of the showcase application were tested on a laptop connected to a power supply with the technical specifications listed in the table 6.3. No other demanding applications were open at the time of testing.

Laptop model	HP Pavilion 14-bf007nc (Windows 10)
GPU	NVIDIA GeForce 940MX (2 GB)
CPU	Intel Core i7-7500U (2.7 GHz)
RAM	2 x 8 GB DDR4, 2400 MHz

Table 6.3: Testing machine technical parameters.

The WebGL application open in the Chrome web browser of version 90.0.4430.93 was running smoothly at an average of 60 FPS, but when the world was being observed by moving in a single direction without stops, islands that were not yet completely generated started appearing on the screen.

The desktop application was also running smoothly at an average of 60 FPS and unfinished islands occasionally appeared in the view of the player but they got finished approximately two times faster than in the browser application.

[7]Results of this thesis in an interactive application: https://www.karavasilev.com/projects/thesis/

Extras & Interesting Features

As a bonus feature, each island type can be assigned a text file containing names that get pseudo-randomly assigned to the generated islands of that type. The example tropical island type was assigned a list of over a 1000 names of real Caribbean islands. These names were collected by a custom made script for scraping a Wikipedia page[8]. The name of each island such as „Morant Cays" is displayed in front of it as seen in the figure 6.4.

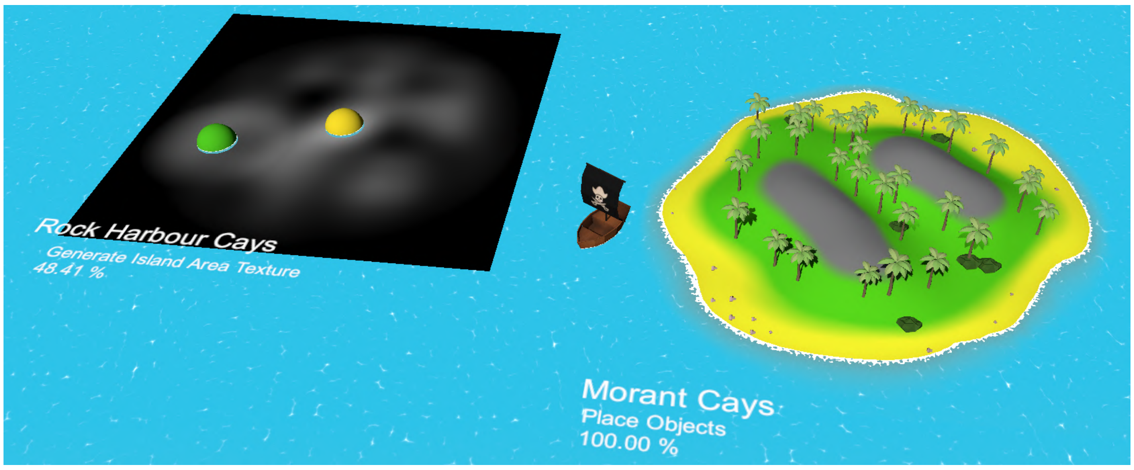
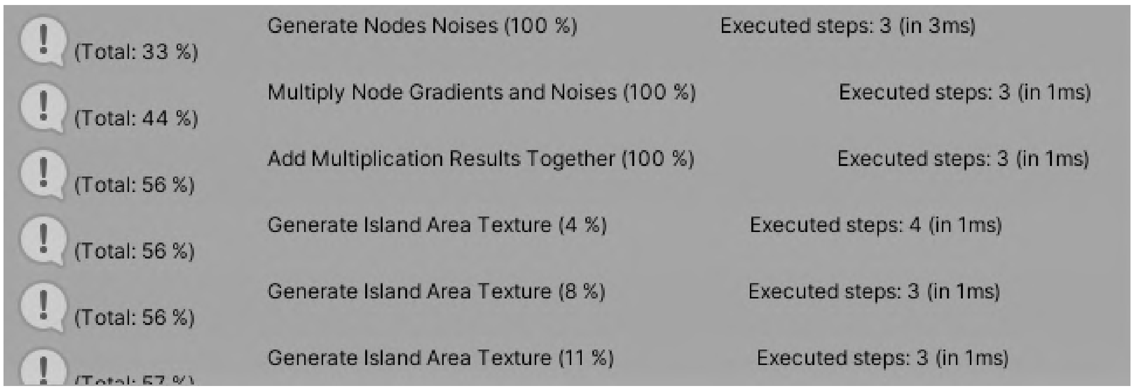


Figure 6.4: Screenshot displaying island names and generation progress.

Another special feature is the visualisation of the individual steps in the process of island generation. It was very helpful during development to be able to visualise what is happening in the code. This feature was used to take some the screenshots for explanations of how the generator is designed and could be used for education purposes as well.

Speaking of debugging and visualisation, an indicator of progress and the name of the currently executed task can be displayed in front of each island as seen in the figure 6.4 right underneath the name of each island. A more detailed logging system capable of measuring the execution times of individual generation steps has been implemented too and an example of its output is shown in the figure 6.5.



```
(Total: 33 %)   Generate Nodes Noises (100 %)                  Executed steps: 3 (in 3ms)
(Total: 44 %)   Multiply Node Gradients and Noises (100 %)     Executed steps: 3 (in 1ms)
(Total: 56 %)   Add Multiplication Results Together (100 %)    Executed steps: 3 (in 1ms)
(Total: 56 %)   Generate Island Area Texture (4 %)             Executed steps: 4 (in 1ms)
(Total: 56 %)   Generate Island Area Texture (8 %)             Executed steps: 3 (in 1ms)
(Total: 57 %)   Generate Island Area Texture (11 %)            Executed steps: 3 (in 1ms)
```

Figure 6.5: Example output of the logging system used for optimising performance.

[8]List of Caribbean islands: https://en.wikipedia.org/wiki/List_of_Caribbean_islands

Chapter 7

Conclusion

This thesis studied different approaches to procedural generation of terrain and its representation, as well as visualisation of animated water surfaces in virtual scenes. The results of the research were used to design a terrain generator with an animated water surface. Traditional approaches to the partitioning of the world and object placement were modified to best fit the selected niche of tools for generating non-realistic procedural worlds consisting of islands in the sea. Furthermore, an approach for generating a terrain of islands by pseudo-randomly generating points called *terrain nodes* defining the terrain's properties is proposed that has not been found elsewhere.

The proposed terrain generator and water surface shader designs were implemented as reusable components for the game engine Unity. These components were used to create a real-time procedural generator of theoretically infinite worlds composed out of islands in the sea. It supports multiple user definable island types, visualisation of the generation process, and the islands produced by it can be very diverse thanks to the devised method of terrain generation. The generator is capable of running in real-time thanks to the developed task system spreading the computational load across multiple frames.

A showcase application for both web and desktop platforms was created to demonstrate an example of a generated world. A demonstration video showing the application on both of the platforms can be found on the attached medium. This tool was found to be able to compete with similar existing island generators available for Unity.

Before publishing the tool on the Unity's store I would like to add more supported types of noise used for terrain generation and improve the performance by converting the most computation heavy tasks such as the texture generation tasks to be executable in parallel.

Other possibilities for future development could be modifying the object placement method to prevent the possibility of object collisions, consider steepness of the terrain during the generation of textures, and extending the object placement method to support placement of objects which need modification of the terrain underneath them.

Bibliography

[1] Perlin, K. An Image Synthesizer. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. july 1985, vol. 19, no. 3, p. 287–296. DOI: 10.1145/325165.325247. ISSN 0097-8930. Available at: https://doi.org/10.1145/325165.325247.

[2] Perlin, K. Improving Noise. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 2002, p. 681–682. SIGGRAPH '02. DOI: 10.1145/566570.566636. ISBN 1581135211. Available at: https://doi.org/10.1145/566570.566636.

[3] Shaker, N., Togelius, J. and Nelson, M. J. *Procedural Content Generation in Games*. Cham: Springer International Publishing, 2016. Computational Synthesis and Creative Systems. ISBN 978-3-319-42714-0.

[4] Cook, R. L. Stochastic Sampling in Computer Graphics. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. january 1986, vol. 5, no. 1, p. 51–72. DOI: 10.1145/7529.8927. ISSN 0730-0301. Available at: https://doi.org/10.1145/7529.8927.

[5] Bridson, R. and Müller Fischer, M. Fluid Simulation: SIGGRAPH 2007 Course Notes. In: *ACM SIGGRAPH 2007 Courses*. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–81. SIGGRAPH '07. DOI: 10.1145/1281500.1281681. ISBN 9781450318235. Available at: https://doi.org/10.1145/1281500.1281681.

[6] Gustavson, S. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*. 2005. Available at: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[7] Johansen, R. S. *A Primer on Repeatable Random Numbers* [online]. 2015 [cit. 2021-05-09]. Available at: https://blogs.unity3d.com/2015/01/07/a-primer-on-repeatable-random-numbers/.

[8] Duncan, J. *Caribbean Sea Islands* [online]. 2019 [cit. 2021-03-06]. Available at: https://unsplash.com/photos/nRZoLSr0mEE.

[9] Elcanetay. *Low Poly Nature - Asset* [online]. 2018 [cit. 2021-05-08]. Available at: https://assetstore.unity.com/packages/3d/vegetation/low-poly-nature-free-vegetation-134006.

[10] Verma, R. *Lowpoly Paper Boat - Asset* [online]. 2016 [cit. 2021-05-08]. Available at: https://assetstore.unity.com/packages/3d/lowpoly-paper-boat-61369.

[11] Broken Vector. *Low Poly Rock Pack - Asset* [online]. 2018 [cit. 2021-05-08]. Available at: https://assetstore.unity.com/packages/3d/environments/low-poly-rock-pack-57874.

[12] Seok Ju, Y. *Pure C# implementation of xxHash* [online]. 2017 [cit. 2021-04-03]. Available at: https://github.com/noricube/xxHashSharp/blob/master/xxHashSharp/xxHash.cs.

[13] Pahunov, D. *MapMagic World Generator* [online]. 2021 [cit. 2021-05-06]. Available at: https://assetstore.unity.com/packages/tools/terrain/mapmagic-world-generator-56762.

[14] Novakova, K. *Tropical Island Low-poly 3D model* [online]. 2020 [cit. 2021-03-06]. Available at: https://www.cgtrader.com/3d-models/exterior/landscape/tropical-island-4a052743-904f-4b36-81a3-99fdc98d9015.

[15] Unity Technologies. *Perlin Noise Implementation* [online]. 2021 [cit. 2021-04-30]. Available at: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html.

[16] 2K. *Sid Meier's Pirates!* [online]. 2021 [cit. 2021-03-06]. Available at: https://2k.com/en-US/game/sid-meiers-pirates/.

[17] Worlds, P. *Gaia Pro - Terrain Scene Generator* [online]. 2020 [cit. 2021-03-06]. Available at: https://assetstore.unity.com/packages/tools/terrain/gaia-pro-terrain-scene-generator-155852.

[18] CodeMonky. *Procedural Island Basis Generator* [online]. 2018 [cit. 2021-05-06]. Available at: https://assetstore.unity.com/packages/tools/terrain/procedural-island-basis-generator-58951.

[19] Exenerate. *Procedural Island Generator* [online]. 2019 [cit. 2021-05-06]. Available at: https://assetstore.unity.com/packages/3d/environments/landscapes/procedural-island-generator-150104.

[20] SegFault Games. *Simple Bezier Island Generator* [online]. 2018 [cit. 2021-05-06]. Available at: https://assetstore.unity.com/packages/tools/modeling/simple-bezier-island-generator-104743.

[21] Ada_King. *Free Trees - Asset* [online]. 2017 [cit. 2021-05-08]. Available at: https://assetstore.unity.com/packages/3d/vegetation/trees/free-trees-103208.

[22] Unity Technologies. *System requirements for Unity 2021.1* [online]. 2021 [cit. 2021-04-26]. Available at: https://docs.unity3d.com/2021.1/Documentation/Manual/system-requirements.html#web.

Appendix A

Island Generation Tasks Class Diagram

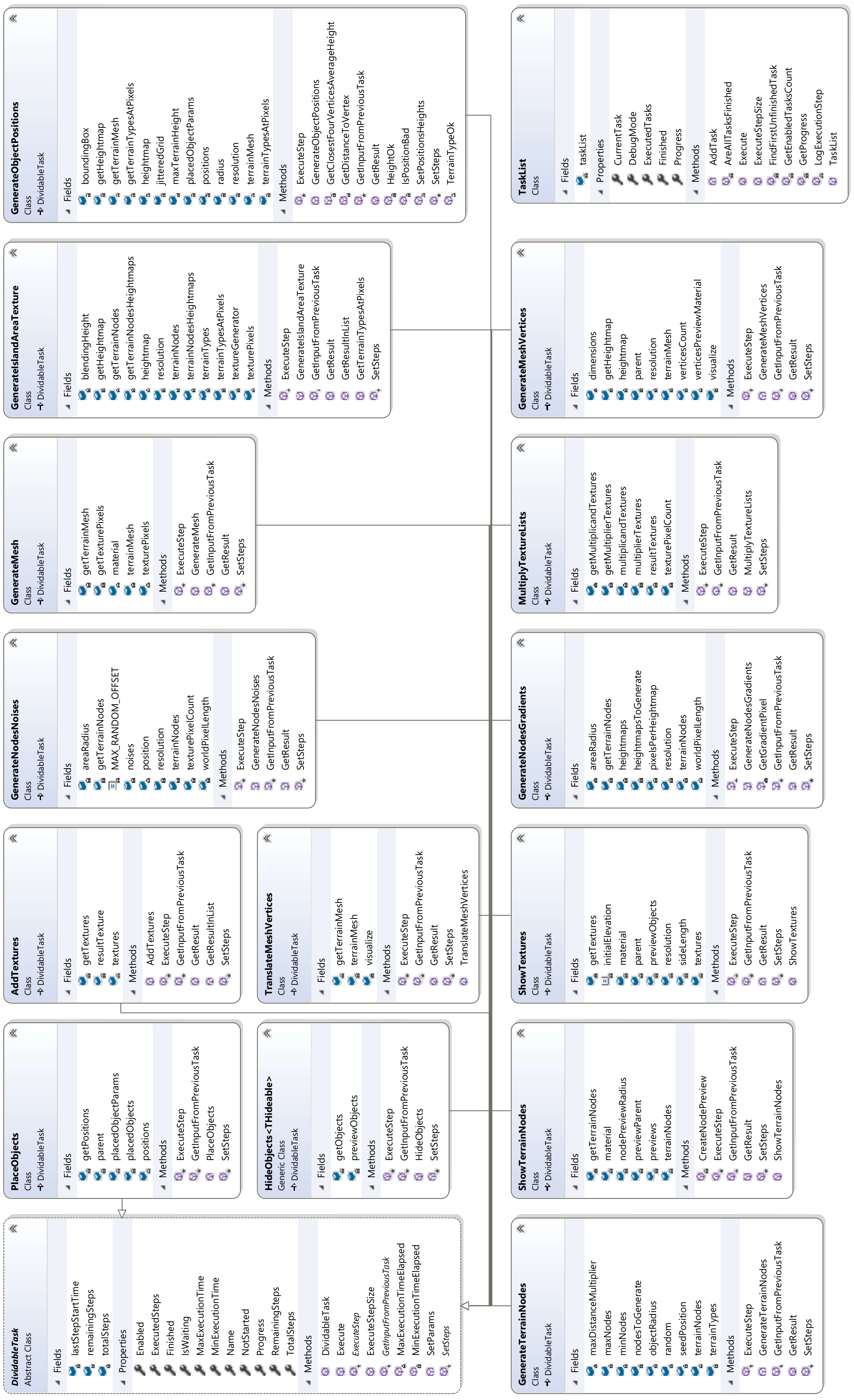
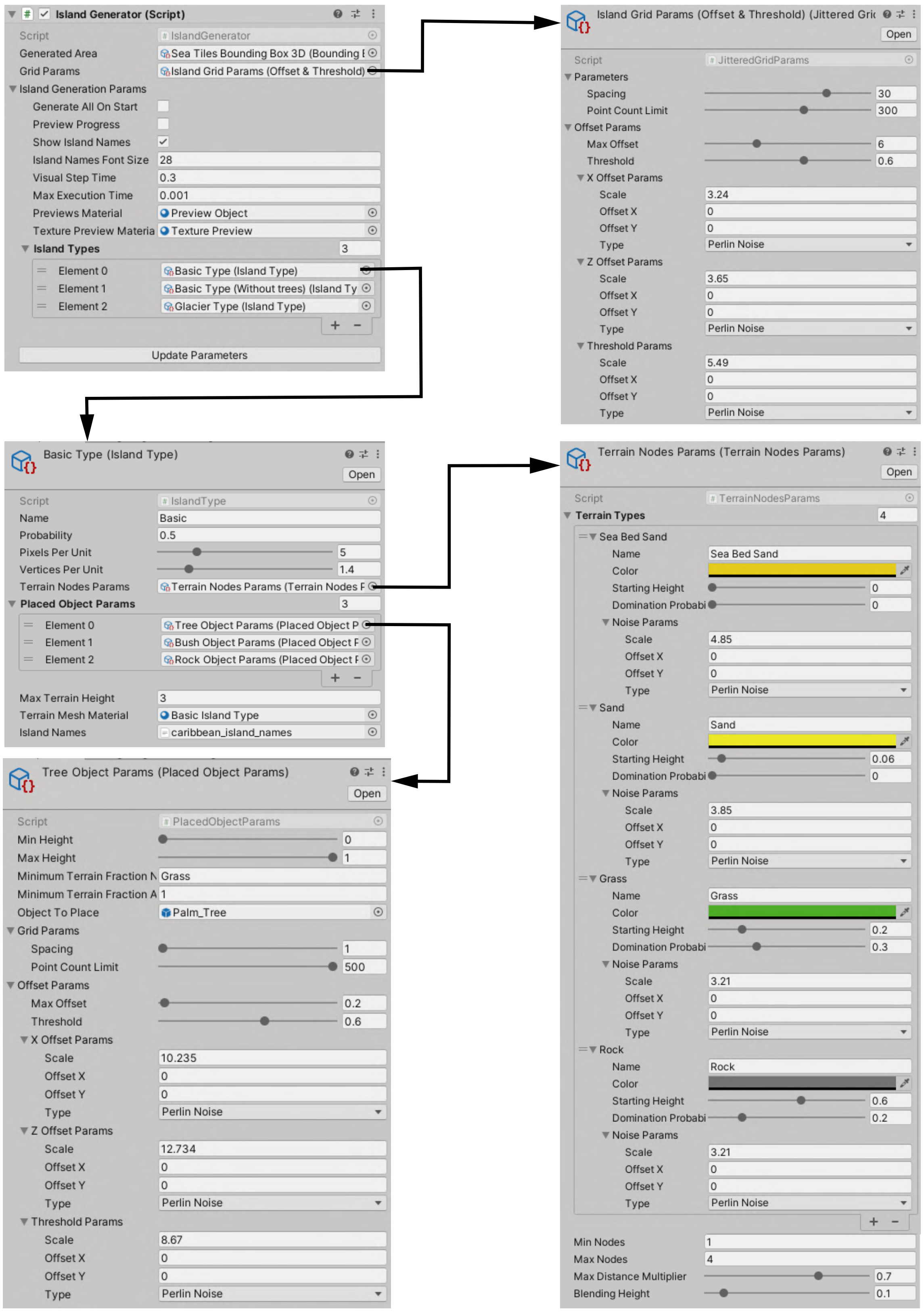


Figure A.1: Class diagram of the dividable task, its decedents and the task list.

Appendix B

Island Generation Parameters



Island Generator (Script)

Script	IslandGenerator
Generated Area	Sea Tiles Bounding Box 3D (Bounding I
Grid Params	Island Grid Params (Offset & Threshold)
Island Generation Params	
Generate All On Start	☐
Preview Progress	☐
Show Island Names	✓
Island Names Font Size	28
Visual Step Time	0.3
Max Execution Time	0.001
Previews Material	Preview Object
Texture Preview Materia	Texture Preview
Island Types	3
Element 0	Basic Type (Island Type)
Element 1	Basic Type (Without trees) (Island Ty
Element 2	Glacier Type (Island Type)

Update Parameters

Island Grid Params (Offset & Threshold) (Jittered Gri

Open

Script	JitteredGridParams
Parameters	
Spacing	30
Point Count Limit	300
Offset Params	
Max Offset	6
Threshold	0.6
X Offset Params	
Scale	3.24
Offset X	0
Offset Y	0
Type	Perlin Noise
Z Offset Params	
Scale	3.65
Offset X	0
Offset Y	0
Type	Perlin Noise
Threshold Params	
Scale	5.49
Offset X	0
Offset Y	0
Type	Perlin Noise

Basic Type (Island Type)

Open

Script	IslandType
Name	Basic
Probability	0.5
Pixels Per Unit	5
Vertices Per Unit	1.4
Terrain Nodes Params	Terrain Nodes Params (Terrain Nodes F
Placed Object Params	3
Element 0	Tree Object Params (Placed Object P
Element 1	Bush Object Params (Placed Object F
Element 2	Rock Object Params (Placed Object F
Max Terrain Height	3
Terrain Mesh Material	Basic Island Type
Island Names	caribbean_island_names

Tree Object Params (Placed Object Params)

Open

Script	PlacedObjectParams
Min Height	0
Max Height	1
Minimum Terrain Fraction N	Grass
Minimum Terrain Fraction A	1
Object To Place	Palm_Tree
Grid Params	
Spacing	1
Point Count Limit	500
Offset Params	
Max Offset	0.2
Threshold	0.6
X Offset Params	
Scale	10.235
Offset X	0
Offset Y	0
Type	Perlin Noise
Z Offset Params	
Scale	12.734
Offset X	0
Offset Y	0
Type	Perlin Noise
Threshold Params	
Scale	8.67
Offset X	0
Offset Y	0
Type	Perlin Noise

Terrain Nodes Params (Terrain Nodes Params)

Open

Script	TerrainNodesParams
Terrain Types	4
Sea Bed Sand	
Name	Sea Bed Sand
Color	🟨
Starting Height	0
Domination Probabi	0
Noise Params	
Scale	4.85
Offset X	0
Offset Y	0
Type	Perlin Noise
Sand	
Name	Sand
Color	🟨
Starting Height	0.06
Domination Probabi	0
Noise Params	
Scale	3.85
Offset X	0
Offset Y	0
Type	Perlin Noise
Grass	
Name	Grass
Color	🟩
Starting Height	0.2
Domination Probabi	0.3
Noise Params	
Scale	3.21
Offset X	0
Offset Y	0
Type	Perlin Noise
Rock	
Name	Rock
Color	⬛
Starting Height	0.6
Domination Probabi	0.2
Noise Params	
Scale	3.21
Offset X	0
Offset Y	0
Type	Perlin Noise
Min Nodes	1
Max Nodes	4
Max Distance Multiplier	0.7
Blending Height	0.1

Figure B.1: Configurable parameters of the island generator.

Appendix C

Water Surface Shader Graph

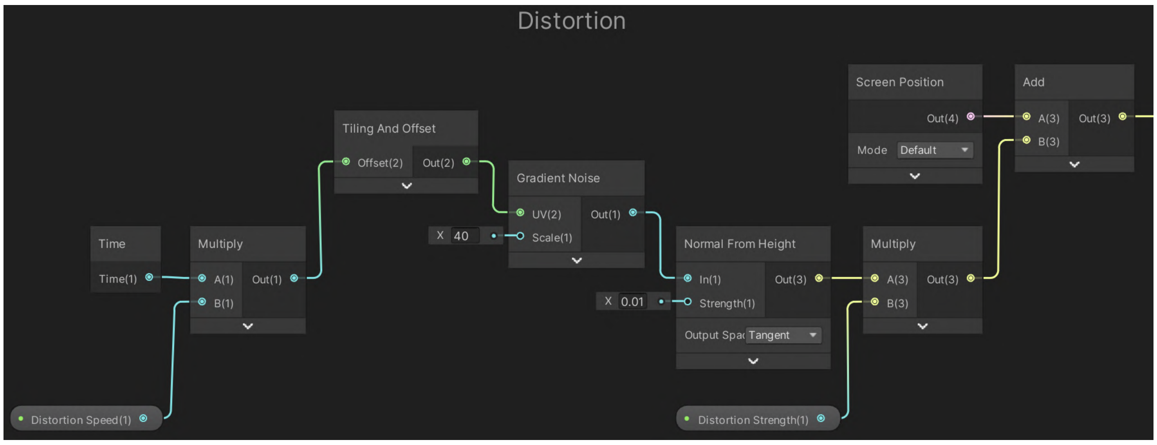


Figure C.1: Distortion effect shader graph group.

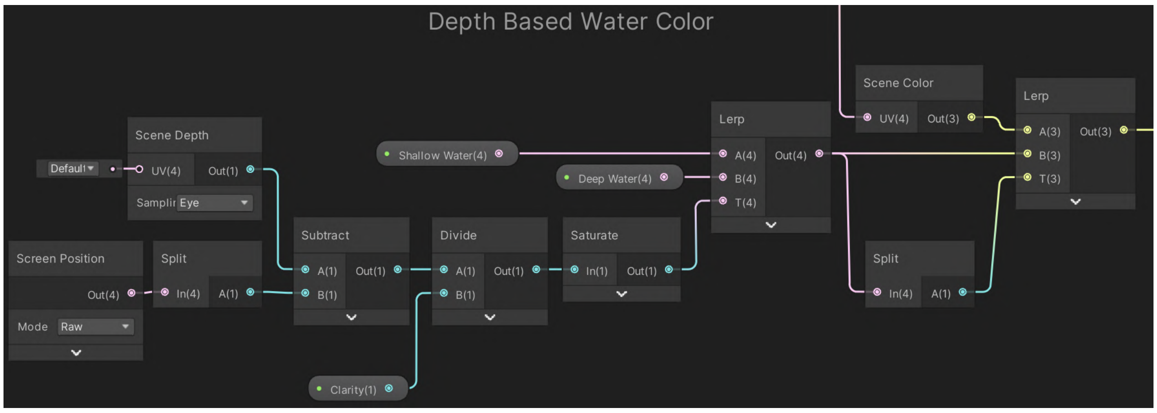


Figure C.2: Depth based water colour shader graph group.

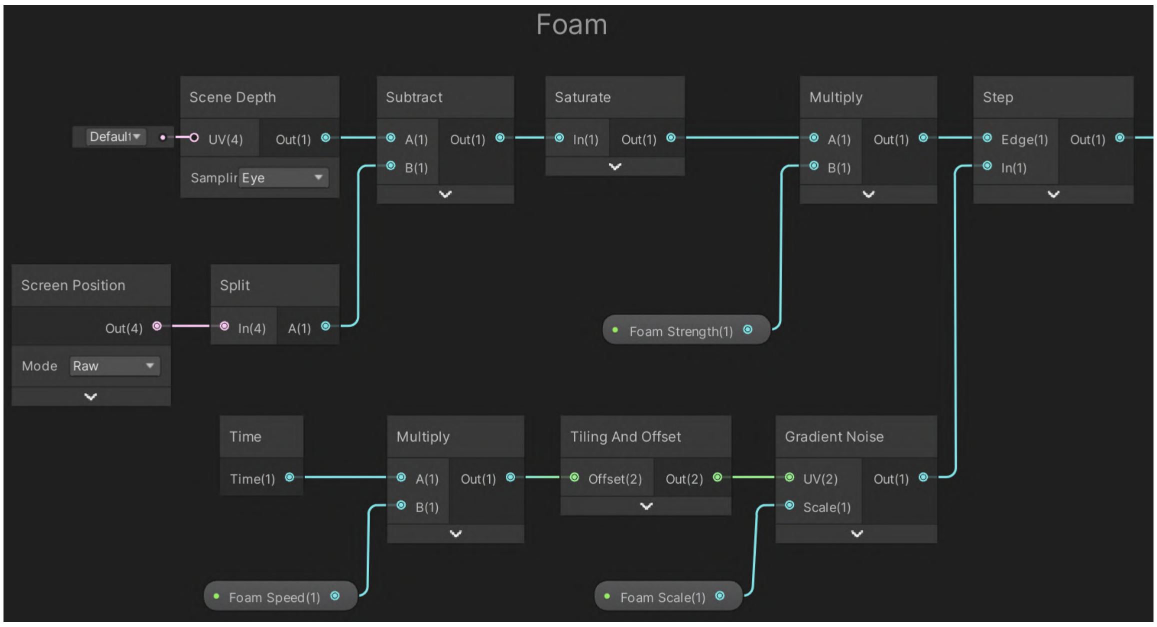


Figure C.3: Foam effect shader graph group.

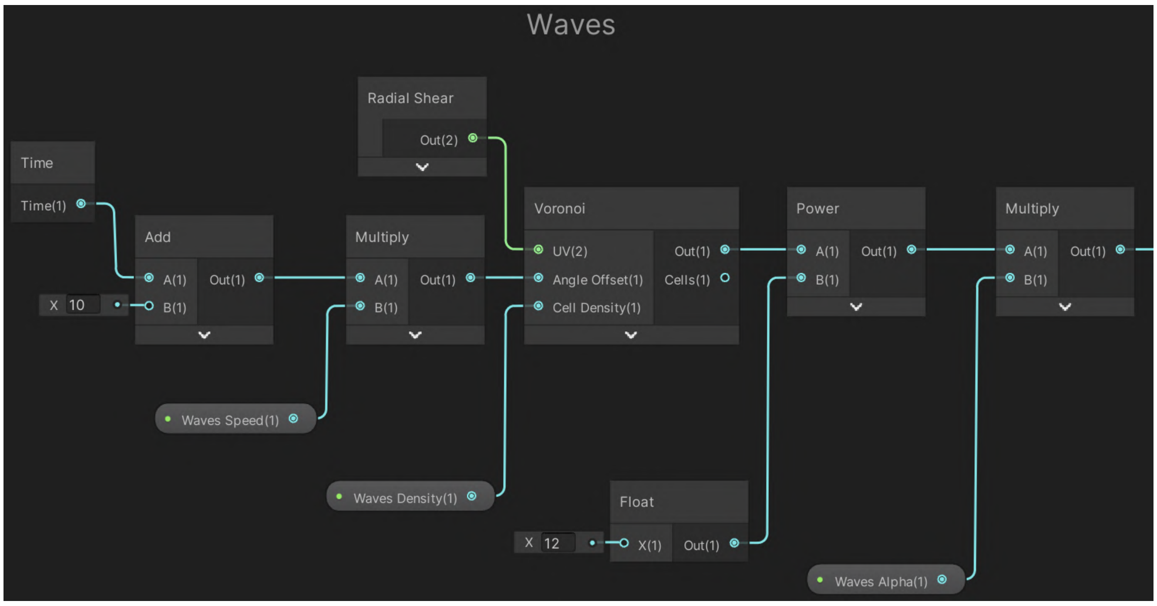


Figure C.4: Foam effect shader graph group.

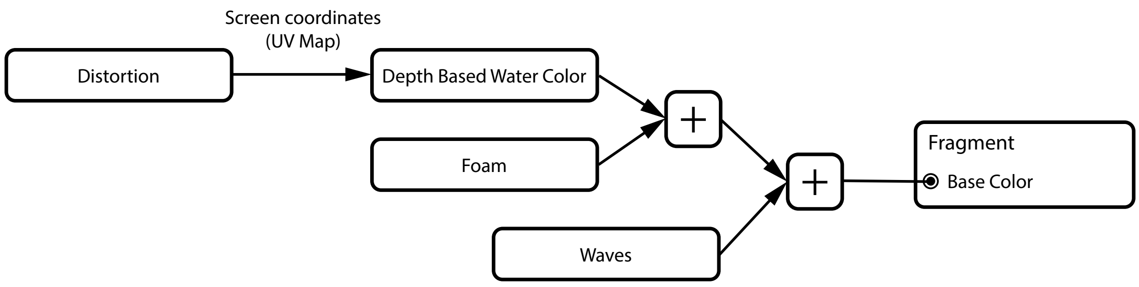


Figure C.5: Shader graph groups connection diagram.