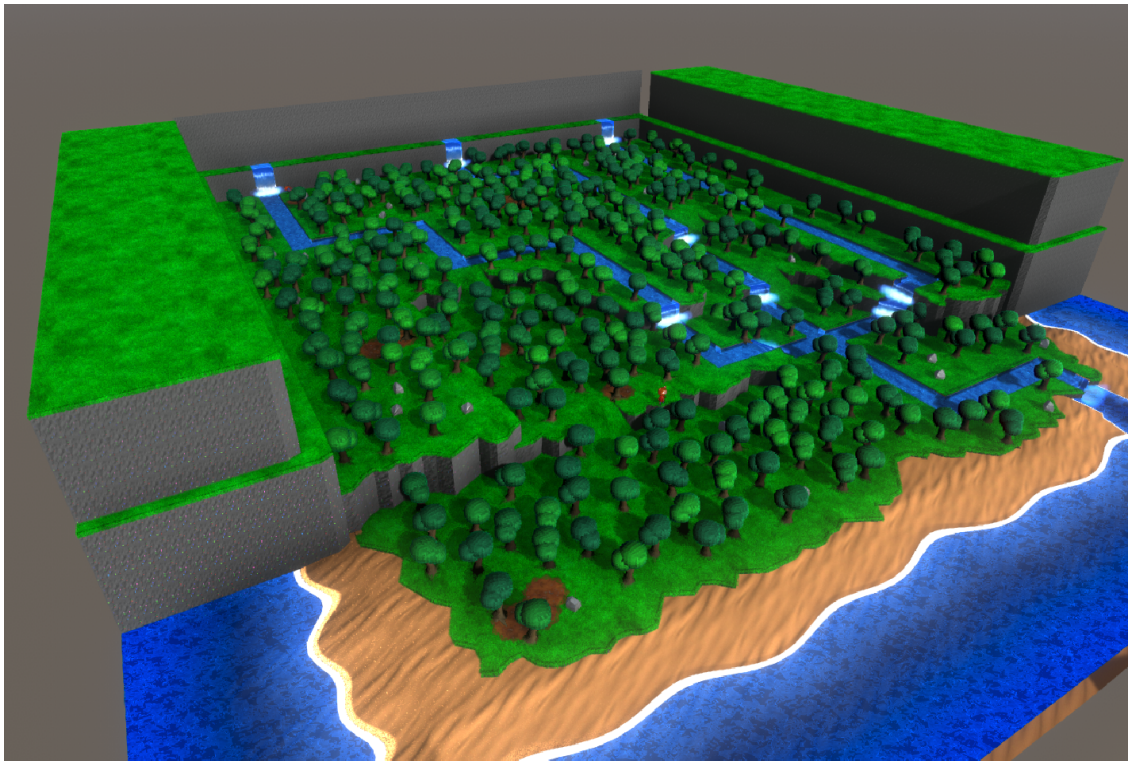

Procedurally Generating Worlds in the Style of Nintendo's Animal Crossing

Master's thesis in Computer science and engineering

Anton Annlöv

Cornelis Törnquist Sjöbeck

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Procedurally Generating Worlds in the Style of Nintendo's Animal Crossing

Anton Annlöv
Cornelis Törnquist Sjöbeck

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Procedurally Generating Worlds in the Style of Nintendo's Animal Crossing
Anton Annlöv
Cornelis Törnquist Sjöbeck

Supervisor: Pauline Belford, Department of Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Example output of the procedural generator.

Typeset in LaTeX
Gothenburg, Sweden 2022

Procedurally Generating Worlds in the Style of Nintendo's Animal Crossing
Anton Annlöv
Cornelis Törnquist Sjöbeck
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This is an investigation to answer the question "How can you generate an Animal Crossing style terrain?". Our aim has been to investigate the challenges and opportunities in terms of solving this task, while adhering to the specific style of Nintendo's Animal Crossing. We recognize the importance of world-building and its' widespread employment in many commercial and research environments. In some cases it is favorable to generate a world as opposed to manually building it. With our investigation we are specifically targeting real-time applications, such as video games. In terms of procedurally generating worlds specifically in the style of Animal Crossing, little previous work outside of the games has been done. Our investigation is taking a step to fill that gap in the research. We approach the problem by combining a collection of know techniques, such as: heightmaps, Lindenmayer-systems, Depth-First Search, Poisson disk sampling, Simplex noise. We also combine them with our own solutions and introduce custom stylized visual elements, like graphical shaders, to shape the aesthetic. Our shaders use techniques like layered textures, displacement maps and scrolling textures.

At the start of this paper we ask the question: "How can you generate an Animal Crossing style terrain?". We answer this question defining what the Animal Crossing style is and by showing how to apply specific techniques in order to produce such a result. By this we answer a number of things: Yes, you can procedurally generate terrain in the style of Animal Crossing; Yes, these worlds can be made suitable for realtime applications such as a game; Applying specific techniques in a certain way combined with some extra details, you can achieve these types of visual results which we believe follow the principles of the Animal Crossing style. Auxiliary to this we also provide insight into ways in which certain aesthetic features can be achieved, which may fit into a world following the style of Animal Crossing terrain.

Keywords: Computer, science, computer science, engineering, project, thesis.

Acknowledgements

Thank you Pauline Belford for undertaking the role as supervisor for our project. Your dedication and eagerness to help us, has been essential for our motivation throughout the thesis. We are very grateful for the time you have given us, and we feel lucky to have had you as our supervisor.

We would also like to thank Pedro Petersen Moura Trancoso, for guiding us through the different steps of the thesis process and our master studies.

Additionally, a big thanks goes out to the Chalmers computer graphics team, and in particular Roc Ramon Currius, for helping us solve a problem with artifacts when generating noise textures.

Anton Annlöv, Gothenburg, August 2022
Cornelis Törnquist Sjöbeck, Gothenburg, August 2022

Contents

List of Figures

List of Tables

Glossary

Depth-First Search A linear graph search method, where you prioritize depth before breadth..

Heightmap A 2D-array of data used to determine height offset for polygon terrain in 3D..

Lindenmayer-system (L-System) A specific method to produce a string of tokens given simple rules..

Poisson Disk Sampling A technique to sample evenly distributed random points, as opposed to completely randomly (not evenly distributed). .

RGB Stands for red, green, blue. The three color channels that make up the final color of a pixel.

RGBA Same as RGB, but with an additional alpha channel for transparency effects.

Shader A shader is a program that runs on the GPU of the computer. A shader program is divided into multiple steps, most importantly when doing rendering work it has a per vertex stage and a per fragment (pixel) stage.

Simplex Noise A technique used to create noise functions. It is more efficient than Perlin Noise[5], which is used in the same way..

1
Introduction

Animal Crossing is a game by Nintendo, containing unique worlds with a distinct associated style. These worlds are procedurally generated at a high level, in that they only make use of larger handmade terrain features that are puzzled together to create variation. The novel contribution of this thesis is to investigate what makes Animal Crossing worlds distinct, and the procedural generation of worlds capturing that distinct style at a lower (finer) level. This has, to our knowledge, never been done before with the combination of the proposed techniques; these techniques being heightmaps, Lindenmayer-systems, Depth-First Search, Poisson disk sampling, Simplex noise, and more, combined with our own solutions.

The creation of virtual worlds is something that occurs frequently within the games and movie industries. What separates a virtual world from other world building is that it must not just be visualized in the mind, but rather be a concrete description visible physically on a screen.

A world is more than just the physical, it also includes the ideas, cultures, and characters that inhabit it. For this work however, we will focus on the more concretely visible aspects such as the terrain and water.

There are different methods to create a world. When it comes to creating its physical form, often a team of artists will sit down and model each individual piece and manually place them throughout the world in whatever tool they are using. There is however a different approach, instead one might employ algorithms to generate or automate certain processes in the creation work. This is called procedural generation.

1.1 Thesis Rationale

The research question being asked is:

How can you generate an Animal Crossing style terrain?

A large part of the games industry is dedicated to the creation of 3D environments. Thousands of artists and programmers are each year faced with how to create a world for their game that is both visually interesting and holistically satisfactory. It is therefore evident that research into creation of such worlds is a worthwhile

endeavour.

The thesis aims to help developers that want to create a world for their game that takes inspiration from the distinct Animal Crossing style outlined. This can come in the form of providing concrete implementations that directly achieve what is desired, or techniques and methods that help inspire other ways of getting the result that is wanted. It also helps explore approaches that may have been considered without having to do the costly time investment to evaluate their benefits and eventual drawbacks.

In conclusion, the novel contribution of the work is the specific combination of known and new techniques used to achieve the results of generating Animal Crossing style worlds and introducing custom stylized visual elements, such as graphical shaders, to shape the aesthetic. The techniques utilized include heightmaps, Lindenmayer-systems, Depth-First Search, Poisson disk sampling, Simplex noise, and more, combined with our own solutions.

1.2 What is procedural generation

Procedural generation is simply the creation of data by computers [6]. This means that any algorithm that generates new data can technically be considered as procedural generation. What is important to take away from this is the fact that there is no one special technique for procedural generation, its an unlimited problem space. The term is often referenced within the video games and movies industries as a lot of content, such as landscapes, objects, characters, animations, and more, can be created procedurally either in its entirety or in part.

Though procedural generation can be used when creating a single specific piece of content, in the case where constructing it through an algorithm might be easier than by hand, what makes procedural generation an attractive option for content creation is the ability to create a lot of unique possibilities from the same algorithm while the workload from humans is small in relation to the amount of content that is created.

A notable example of procedural generation in video games is the game No Man's Sky which was released in 2016. The game features an explorable universe where each planet, its landscape, vegetation, and animal life, among other things, is all generated content. This made it possible to provide players with access to over 18 quintillion unique planets which they can discover and explore[7].

The release of No Man's Sky also highlighted some of the challenges that procedural generation brings. Even though the game was able to generate so many different worlds, many people noted that a lot of the worlds didn't feel distinct enough from each other. There were also comments regarding the blandness of many worlds or just outright examples of generated content that, in their opinion, wasn't very

good.[8]

This is a challenge widely discussed with procedural generation. It can be hard to create an algorithm that produces the quality of handmade work while retaining the ability to produce a large amount of content with sufficient variation.

There are other aspects of procedural generation which makes it interesting. By generating content at runtime, not to say that all procedural generation is done at runtime, there exists the possibility to have a reduced memory footprint on the hard drive since the content does not exist until it is needed. The down side of this is that generating content at runtime is likely to increase load times because of the assets need to be created rather than just read from disk.

1.3 An Animal Crossing style world

The thesis aims to investigate the generation of not just any world, but a world with a specific and distinct style. That style being taken from the Nintendo game series Animal Crossing. The question then follows: What is an Animal Crossing style world?

The following description is our own, as it is not defined in an exact way in any literature we have discovered. The description is extrapolated from our own observations and therefore subjective in nature. This means that a different party may come to a different description given an independent study, however for the purpose of the thesis all work will presume the description given here as the definitive.

First and foremost it is important to note that Animal Crossing is a franchise with, at the time of writing, 5 titles considered main series, if only including worldwide releases. The first game Animal Crossing: Population Growing was released in 2001 and the latest entry Animal Crossing: New Horizons was released 19 years later in 2020. Over these 19 years the core of the game and its aesthetics has remained similar but not static. Technological advancement and new design choices has led each entry to leave its distinct mark.

1.3.1 Cliffs

The layout of an Animal Crossing world, in terms of its terrain, that makes it so distinct comes down to a couple factors. The most divisive being that the terrain is divided up into layers separated by cliffs. Each layer in itself is completely flat, with exception to slopes which connect two neighbouring layers creating a traversable path between them. Figure 1.1 shows a map view of the first Animal Crossing game, where a cliff is clearly visible dividing up the map into two layers. The number of layers has varied from title to title as can be seen in table 1.1.

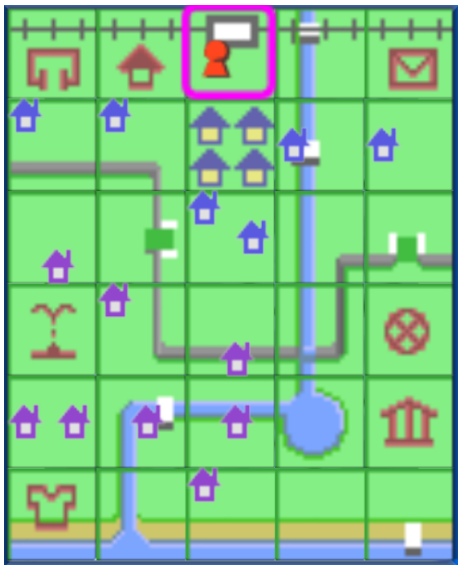

Figure 1.1: Map from Animal Crossing: Population Growing (2001). Two terrain layers separated by a gray line can be seen with slopes connecting them together. Here the map can also be seen as divided up into a grid, each square representing an acre.

Table 1.1: Number of layers in the terrain for each game [4].

Animal Crossing: Population Growing (2001)	2 - 3
Animal Crossing: Wild World (2005)	1
Animal Crossing: City Folk (2008)	2
Animal Crossing: New Leaf (2012)	1
Animal Crossing: New Horizon (2020)	1 - 4

A distinct feature of the Animal Crossing style is the camera placement. The camera views the player from a top down view at a slight angle facing north at all times. This unique placement of the camera has undoubtedly influenced the formation of the terrain. Excluding the latest entry, which we will discuss later, all worlds are designed in a staircase formation in terms of the terrains layers. In other words, going from north to south on the map you will never traverse upwards in elevation only down. This was most likely settled on due to the fact that having terrain be at a higher elevation to the south, in the cameras direction, would result in the view of the player character getting obstructed by terrain. At the same time it resulted in an iconic terrain style, see figure 1.2 for example of camera view in game.

Figure 1.2: Example view from Animal Crossing: Population Growing.

The worlds are divided into acres, see figure 1.1 for example. These acres are uniformly sized and work as meta building blocks for the worlds construction. An acre can either be completely part of a single layer, and therefore not have any cliffs in it, or contain a cliff splitting it between two layers. An important constraint here is that a cliff line is always completely enclosed inside an acre, meaning it does not curve freely back and forth along its major axis between two different acres.

As can be deduced from the previous paragraph: cliffs are not completely straight. Excluding the last entry again, which will be discussed further down, cliffs move in a squiggly fashion which are best understood by looking at images, see figure 1.3. As can be viewed in the figures, the cliffs have quite a free form motion to them.

Figure 1.3: Animal Crossing: Population Growing, examples of cliff formations.

Figure 1.4: Animal Crossing: Population Growing, example of world border cliff in left image and ocean beach view in right image.

Starting from the first game, Population Growing, one aspect of the terrain that has stayed is the beaches. The acres on the south edge of the world contain a beach bordering the ocean while the acres on the eastern and western edges of the world contain high cliffs acting as walls that the player can not pass, see figure 1.4.

There are a couple of caveats that should be mentioned for each title. The first game Population Growing will most of the time generate a 2 layered map, but can generate 3 layer ones. Wild World is a single layer world, meaning no cliffs. City Folks went back to the 2 layered approach without the possibility of 3 layers. New Leaf then followed up by going back to a single layer except for when going to the beach which was separated by a cliff from the rest of the map. It also introduced that one of the eastern or western edges of the map now also acts as a beach together with the southern edge. Finally, we have New Horizons which has made the most distinct deviation in terms of terrain formation.

The game features, for the first time in the series, modifiable terrain. This new addition has led to certain design constraints not present in the older titles. The worlds now spawn with 3 layers, but due to the player's ability to modify the terrain a map can be moulded into having anywhere between 1 and 4 layers. The topmost, fourth layer, is however not traversable by the player.

Another distinct change that the modifiable terrain has brought with it is the formation of the cliffs. In earlier titles, cliffs would have a much more free flowing structure to them, but now, in order to facilitate the ability to increase or lower the elevation at any tile, all cliffs are either completely straight or angled at 45° degrees which makes them easy to tile, see figure 1.5. This design decision means greater player freedom in shaping the world, but also introduces a greater homogeneity to

the formation of the cliffs.

Figure 1.5: Animal Crossing: New Horizons, examples of cliff formations.

1.3.2 Rivers

The next distinctive feature of Animal Crossing worlds are its rivers. Each world since the first game has always included at least one river. A river is a body of water that flows from the northern edge of the map to the southern edge in a snaking fashion. As with cliffs, they transition from one acre to another when reaching an acre edge, but do not transition back and forth between two acres. This means that a river section is completely continuous within a single acre. In figure 1.1, you can see a map view of the river formation, and in figure 1.6, you can see an example of a river in game.

Figure 1.6: Animal Crossing: Population Growing, example of in game river.

A river may, while in an acre that is entirely contained within a single layer, transition into a pond. A pond is a larger body of water completely contained within a single acre, often containing some set dressing such as lily pads or a small dock.

When a river flows through a cliff, which it intuitively only does downwards, it transitions into a waterfall. As cliffs only go down a single layer at a time, waterfalls likewise always fall the same distance. An exception to this is when a river flows into the ocean, which it does through a waterfall in all but Wild World and New Horizons.

A river may split into two separate branches continuing their own separate journeys to the ocean. When such a split happens, the rivers will exit the acre through different edges. Two rivers will never merge, only split. In all games, excluding the latest entry, there is only ever a single initial river. That river may however as mentioned split exactly once.

Some other minor features related to rivers. Along a river, bridges may be placed in order to allow the player to cross. Usually there is one bridge placed per river per layer. There are small holding ponds that can spawn in the world. These are small bodies of water not connected to any river or other water feature.

The last entry, New Horizons, is an exception to a large part of the description above. In this game, players can increase and decrease elevation at their leisure, and when lowering the elevation level enough they create water holes. A string of water holes becomes a river, meaning the distinction between rivers and lakes has disappeared from this entry. Now rivers and lakes can be of any shape and size, with the exception that the orifice into the ocean is of static size and placement.

In return for the increased freedom in river and lake shape, the water surface no longer has a clear flow direction as it did previously. Instead there, from observation, seems to be a default water effect that is shown whenever a tile is below water surface level that is not affected by a body of water's shape or size. There are however details added at edges to act as flow lines, see figure 1.7.

Figure 1.7: Animal Crossing: New Horizons, example of flow line details added to rivers. The flow lines can be seen when looking at certain corners at the edges of the river.

1.3.3 Trees and Rocks

When it comes to the actual flat plains of a layer, each Animal Crossing game has had its own approach for decoration. Across all games there have been a lot of evenly spaced trees and rocks throughout the world. Other elements include weeds and flowers. In the first game, Population Growing, there were distinct earth and clover leaf-like patches as well. See figure 1.8.

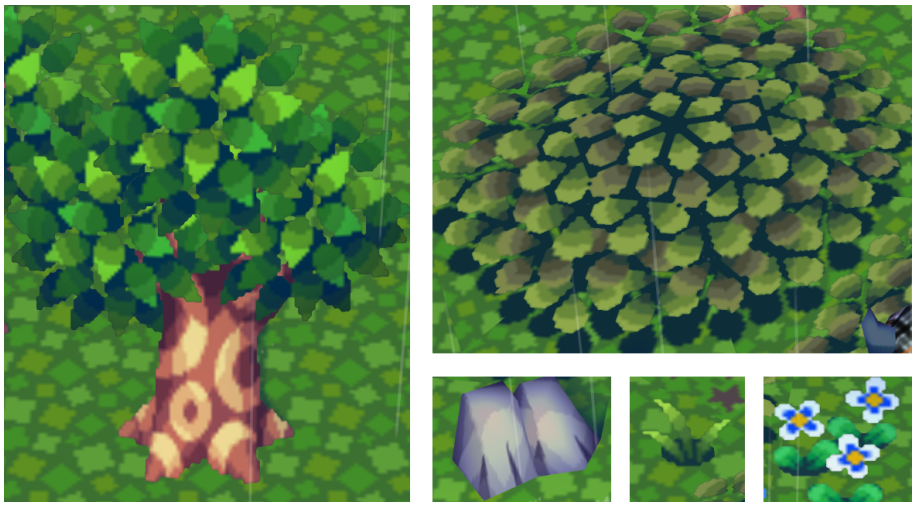

Figure 1.8: Animal Crossing: Population Growing, examples of different terrain decorations. Left is a deciduous tree. Top Right is a clover leaf-like patch. In the Bottom right most left is an example of a rock, middle is an example of a weed, and right is an example of some flowers.

1.3.4 Summary

This thesis work will focus on trying to capture the overall distinct features of an Animal Crossing world, staying true to its distinct style while also finding its own unique takes on the different elements.

The aim is to procedurally generate Animal Crossing worlds more akin to the installments before New Horizons. The cliffs should have a more free form design, not just 45° angles, to them, and the acre layout should be in a staircase-like structure, preventing the terrain from obstructing the cameras view of the player. Between elevation layers will be slopes to traverse between them.

The slopes aim to imitate those found in Population Growing and City folk by being both partly embedded into the cliff and partly creating a mound outside the cliff wall bounds, see figure 1.9. The goal of the slopes is to allow good traversal around the world. This means that no acre should be impossible to reach, while simultaneously not cluttering cliff walls with too many slopes. What too many slopes is, is subjective. However, the definition assumed in the thesis is that each layer island should provide a single slope to all neighbouring layer islands. A layer island is the connected acres of a layer, meaning if a layer has two groups of connected acres, then there are two layer islands on that layer.

Figure 1.9: Animal Crossing: Population Growing, example of slope. Top shows the top of a slope integrated into the cliff. Bottom shows the bottom of a slope and the mound it creates outside the cliff wall bounds.

Rivers should meander and have clear transitions into lakes and waterfalls, as well

as indicate a correctly oriented flow toward the ocean. The rivers shall also have bridges crossing them to allow for traversal into all areas of the world.

There are some aspects where a hybrid approach was chosen for the style. Beaches should generate on the southern edge, while beaches on the western and eastern edges should generate up until the first cliff, after which the edges are covered by a world border cliff. This gives a slight variation to what has been seen in the games thus far while still staying true to their style.

For the tree and rock placement the aim is to have certain zones have more traditional forest generation with many trees, while other areas can be more void of trees creating meadows. There can be the possibility thus to define many different types of biomes with different sets of valid vegetation and other objects.

A goal is for the generator to be flexible in what size the world is and how many levels of elevation are allowed. This will allow the creation of bigger and/or more mountainous worlds with a lot of elevation change, but also more traditional worlds seen in the franchise installments.

A visual aspect of the Animal Crossing games is the curved world look, present in all games but the first. While the main focus is to allow viewing the world in the manner presented in the first game, Population Growing, we also wish to provide the ability to view the world with curvature akin to the later installments.

Finally, in order to showcase the traversability of the world and visual results from an Animal Crossing game point of view, an anthropomorphic player character should be provided that can walk around and interact with the world mesh through collision.

1.4 Evaluation

We evaluate our project from a number of criteria. Firstly, which terrain features from the Animal Crossing games does the generator handle. Also, if there are features in line with the style that are created by the generator not present in the games. Theses features will be evaluated not only on a exists/does not exist basis, but also shown side by side with the equivalent features from the games.

As part of the project, we aim to not only look at function and form, but also graphical style. Because of the terrains non-realistic nature, traditional realistic graphical effects might not be the best way to go. The project will therefore look into possible options of graphical techniques that might be suited well for the world. A certain level of artistic consistency will be aimed for and evaluated through the showcasing of the final terrain.

Another important part of the evaluation is performance. The main inspiration of the project comes from the Animal Crossing games, it is also in games that we

see the projects findings being most applicable. Therefore, it is important that the project can, or at least shows the promise of being able to, perform according to requirements for games. Most aptly, a) the final produced terrain must be able to run in real-time, and b) the generation time can not be too long. The numbers for these two performance criteria will be presented in the results, and also compared to what we define as reasonable.

We define the reasonable run-time performance to be 10 FPS, but should hopefully achieve at least game industry standards of 30, or even 60, FPS. We also define a reasonable load time to be no more than 30 seconds. As we consider this okay, for a one time load. These numbers have been selected by us from self evaluation. The hope is that the generator will be able to perform within theses limits, with good margins. The most important aspect of this evaluation is to evaluate if its even within the realm of possibility to be useful in a real application.

Finally, because of the graphical nature of the project, a part of the result section will showcase a number of complete terrains generated. This is a visual evaluation, where the entire composition of the terrain can be seen and judged from a ”what you see is what you get” perspective.

1.5 How the report is structured

After the introduction a background section will follow. This section will present a literature review of related work and the underlying ideas that are built upon. It is divided up into the main aspects that we have chosen to focus on when generating the virtual world, these being the terrain, grass, water, and tree and rock placement. As well as some sections for other tangent work that was done.

Following the background section is the method section. This section uses the same structure and presents the actual implementation work that was done to achieve the results, which are presented in the next section.

Finally a conclusion is given where we go over our own opinions and ideas on future work that can be done among other things.

2
Background

2.1 Previous work

The following background section presents the previous works and ideas that have laid the foundation for our project. While our thesis has a strong connection to the Animal Crossing games and its terrain, no other papers to our knowledge have tackled this specifically. As such, we do not refer to any previous work that concerns itself with Animal Crossing and its terrain directly. There is however a large amount of research in the area of terrain generation and techniques that can be repurposed for this agenda.

2.2 Textures

When we want to set the color it is generally not feasible to define all the ways in which the the color across an object should be set in the shader program. For that reason, we use textures to define how an object should be colored. We call this pixel shading[9]. Textures give us the freedom to meticulously define details on objects without having to come up with mathematical models in the shader program for doing so. While the most commonly used textures are diffuse textures (a texture which define color), there are a multitude of other ways to utilize textures. In essence the imagination sets the boundaries for what problems you can solve with a texture, however there are some commonly used techniques that are used.

A displacement texture is a black and white texture which is used displace the coordinates on a diffuse texture for example. Black pixels in the displacement map will pull the texture up into the left, and white pixels will push the texture down towards the right, and 50% gray will leave it unchanged [10]. In figure 2.1 you see an example of a displacement texture used for the waterfalls in section 3.4 and section 3.4.1.2. Another way to use a texture is to utilize the RGB channels to provide information about how different parts of an object should be rendered. We show an example of utilization of the RGB channels, through a masking texture, in section 3.3.4.

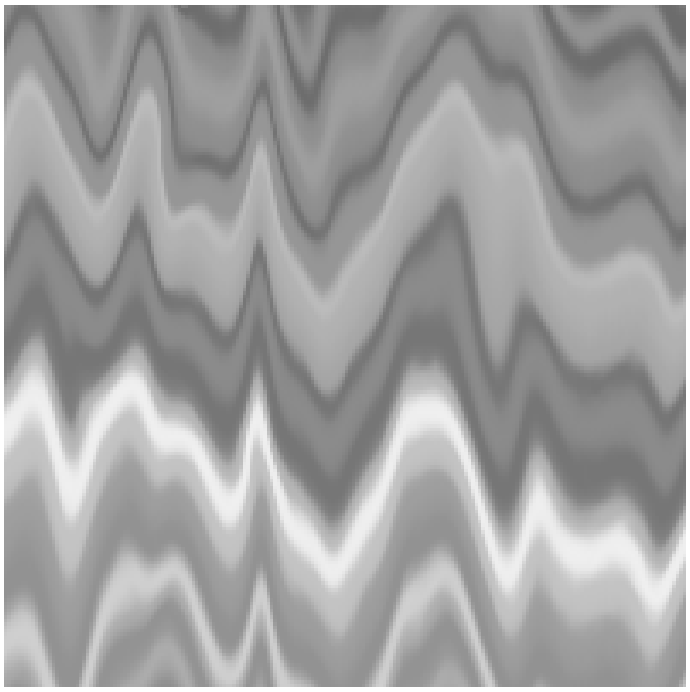

Figure 2.1: Displacement texture used in the waterfall shader program.

2.3 Cliffs and Slopes

In order to understand the reasoning behind the method used to create and render the terrain, it is important to review common techniques used in terrain generation and rendering.

2.3.1 Heightmaps

In computer graphics many techniques make use of textures to store and read data. A texture is a block of data divided up as an array of values. These arrays can be semantically divided into many dimensions, which in computer graphics most often is as a 1D, 2D, or 3D texture. Intuitively, we often think of textures as being semantically divided into two dimensions and contain color data, however in reality color data is just numbers like all other data. This means that what type of data a texture contains is purely semantic and it is up to the user of the data to interpret the numbers as seen fit. Therefore, when textures are discussed in a more general sense, they are often referred to as maps instead. The widespread use of textures as a data structure in computer graphics most likely has to do with their semantic layout being useful for many graphics problems and therefore computer graphics cards have been specialized to read data from them.

Heightmaps are a widespread concept often used when rendering terrain, as can be seen in [11][12]. A heightmap is a 2D texture where the array values are interpreted as the elevation level of a point on the terrain. In other words, a landscape can be viewed as a 2D plane where each point on the plane is offset in the upwards direction

by some value, see figure 2.2.

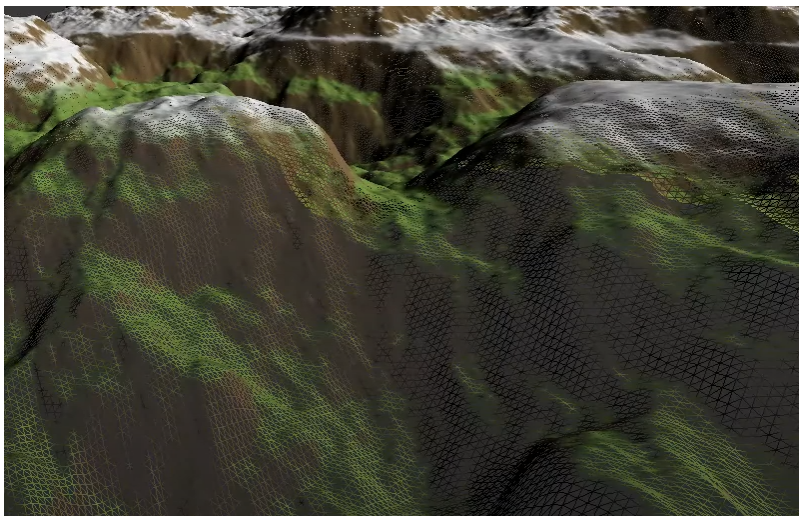

Figure 2.2: A terrain constructed by displacing a connected grid based on a heightmap.

The detail of the terrain is dependent on the resolution of the texture used as a heightmap and the underlying mesh size. In order to achieve greater rendering performance, it is often desirable to have varying levels of detail at different areas of the terrain depending on the current view. For heightmap based terrain rendering there is a large body of research into such techniques.

One example is [11], which divides the terrain into a quadtree (see paper for details) structure dependent on the viewing direction of the camera and its distance from a given point. Each node of the tree is then rendered with the same standardized grid mesh, which results in larger nodes having a lower level of detail than smaller ones. In order to avoid seams in the terrain at node borders with differing resolutions, a morphing technique utilizing the attributes of a quadtree is employed to seamlessly transition from one resolution to the other.

A key aspect of the traditional heightmap based rendering that is necessary to consider is that it is inherent to the technique, unless combined with other methods, that the terrain can not produce overhangs or completely vertical cliffs. This is due to a 2D plane only having a single point at any coordinate, while multiple would be needed for the attributes mentioned. On the other hand, the simplicity of it makes it trivial to implement in its most primitive form.

2.3.2 Voxel Terrain

In order to inherently support terrain features such as overhangs, vertical cliffs, and caves, many employ the use of voxel terrain. A voxel is a data point in an evenly spaced out 3D grid and in the context of terrain is used to represent some feature of the ”box” it is contained in, inside the grid. Voxel based terrain has been privy

to a significant amount of research, and there are a number of different techniques that have been developed [13][14]. Voxel based techniques are about extracting a polygonal mesh from the isosurface (the scalar field referred to as voxels).

One of the first isosurface extraction algorithms was developed in 1995 by W. Lorensen and H. Cline [13], called the marching cubes algorithm. It was not developed initially to be used for terrain rendering specifically, but it is applicable nonetheless. The general idea is to divide a three dimensional space into a grid where each point of the grid contains a scalar, which is what we refer to as the voxel, representing if the point is inside or outside the terrain. A scalar value of one means it exceeds or is on the surface of the terrain, and a value of zero means that it is contained within the terrain.

The algorithm works by identifying that a cube within the grid contains eight voxels, one in each corner. Each voxel can either be inside or outside the terrain, meaning for each box there is a finite number of combinations possible. The paper presents a set of fourteen combinations that account for all possible cases when symmetries are accounted for. Each of these fourteen combinations can then be modeled separately and when inserted into corresponding boxes in the grid, combine to create the polygonal mesh.

The technique is simple compared to many later developed methods, but there are also some aspects worth considering. The static nature of the smaller meshes that combine to create the finished mesh mean that the terrain is prone to certain artifacts. All cubes containing the same voxel combination will look identical, which may not be desirable. Later amendments to the technique allow voxels to be defined as a decimal between zero and one, which is then interpreted as how close a point is to the surface. This is then used to interpolate the edge positions of the smaller meshes, giving the possibility of slight variation in each box. Another artifact present in the finished mesh is that it simply can only create a finite set of shapes, again because of the static nature of its components, meaning certain features are hard or impossible to achieve. For example, a cube with perfect 90° edges or a perfect sphere is impossible to achieve due to the prefabricated smaller meshes shape. By increasing the voxel resolution, the approximated final shape can be more detailed and therefore closer to the intended shape, but this comes at a cubic memory cost due to the three dimensionality of the data.

T. Ju et. al. [14] propose a method for rendering voxel terrain that utilizes hermite data (exact intersection points and normals) tagged to each edge in the voxel grid. This data allows more freedom with the shape of the mesh in each grid ”box”, now perfect cubes with 90° corners are possible for example. The drawback, compared to marching cubes, is that now more data must to be provided, which means more data must be generated when dealing with procedural terrain.

2.3.3 Lindenmayer-Systems

For our Animal Crossing style terrain generator, concepts from both heightmap and voxel based terrain are useful, however they do not seem, by themselves, best suited to achieve the style that is aimed at. A major part of the terrain generation is instead inspired by a different method, called Lindenmayer-Systems (L-Systems), which is a technique found in procedural generation tasks, but maybe not so much when it comes to terrain generation normally.

An L-system consists of: an alphabet, a set of symbols used to create strings, production rules, which state if a symbol can be replaced by one or more symbols, an initial ”axiom”, a string from which the string construction should begin, and a mechanism of translating the generated string into a geometric structure.

An explanation by example would be:

Alphabet: (A, B, c, d)
Production rules: A -> BBc, B -> cd.
Axiom: ”AA”
Result: ”AA” -> ”BBcBBc” -> ”cdcdccdcdc”

The original intent of the work was as a means of describing the behaviour of plant cells and growth processes of plant development. However, In the paper by P. Prusinkiewicz [15] they use L-systems to generate strings which they then show can be interpreted as movement commands for a ”turtle” to walk according to while drawing lines. This shows the potential in using L-Systems to procedural construct graphics related output.

An attribute of L-Systems is that they in each step of the string construction apply the rules to each symbol simultaneously. In the section inspired by how L-Systems work in our terrain generator, it uses more of an iterative approach applying rules to the end of the string. This is more akin to a traditional formal grammar, however the visualization aspects of L-Systems is very apt in how we visualize cliffs, and the concept of stochastic L-Systems, L-Systems with production rules that are chosen based on some probability, is a concept utilized.

2.3.4 Depth-First Search

Another concept connected to our use of L-Systems, is the principle of depth-first search. In his paper, Robert Tarjan [16] presents this method to searching linear graphs. This is not the only method as there are many different methods possible when searching a graph, such as breadth-first [17], monte-carlo tree search [18], or A* [19].

Depth-first search works by, as the name indicates, prioritizing searching vertically before horizontally. Imagine a tree graph, the basic idea is to start searching downwards for some number of moves or until an end node has been reached, before

checking other branches. This stands in direct opposition to breadth-first search which prioritizes searching horizontally before vertically.

2.4 Fur Shells

Looking at all areas of land in the world, about 20-40 percent of it is grass.[20] On a further note, most mammals inhabiting our planet possess hair or fur. Consequently, methods accommodating for this fact are needed when depicting the world in video games, movies, or any other digital models. Here we are going to focus on real-time applications, such as video games.

2.4.1 Different Approaches

A naive approach at making grass would be to geometrically model each grass straw and multiply it over the screen[21]. This just is not feasible for most real-time applications due to the sheer number of grass straws in an average scene. Hair, while possessing many of the same properties leading to the same problems as grass, is sometimes a different story. The main character in a video game is a focal point for the player, so directing resources at achieving geometrical intractable hair can be motivated. Moreover, most scenes housing hair and grass, would theoretically contain undeniably more grass straws than hair strands. This, together with the fact that grass normally is not a central part of most video games, and requires minimal interactivity, is enough inducement to visualize grass in alternative less expensive ways, for most real-time applications.

The most simple approach at visualizing grass is to apply a texture to the ground mesh. This used to be the only option in the infancy of 3D graphics, as resources where already very limited. See figure 2.3 for an example of this.

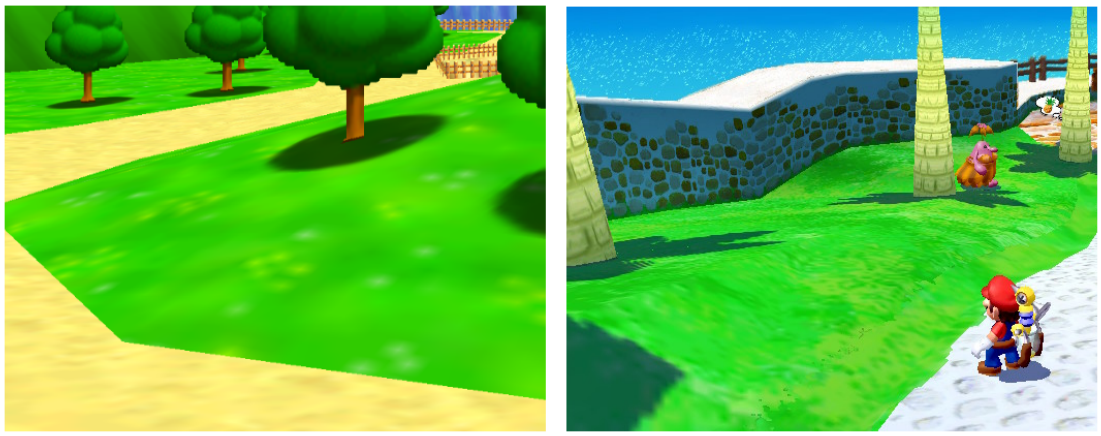

Figure 2.3: Example of applying a texture to the ground mesh to visualize grass from Nintendo's Super Mario 64 (1996), and Super Mario Sunshine (2002) respectively.

The subsequent way to add more complexity to grass, without adding that much more cost to performance, is combining semitransparent 2D textures with the ground texture. An example of this would be Nintendo's The Legend of Zelda: Wind Waker (as seen in Figure 2.4). Wind Waker is a bit of an interesting edge case example however, as patches of tall grass actually play a part in the gameplay. As such, they are their own mesh, and the other grass that the player cannot interact with is just a texture on the ground mesh. This method (namely combining the usage of semitransparent 2D textures and a texture applied to the ground mesh) is widely employed even today, with an increased number of meshes and higher quality textures. See Figure 2.5 for a modern example of this in Ubisoft's Far Cry 6, and a more stylized modern example of this in Nintendo's Animal Crossing: New Horizons. A related method when creating hair (as presented in [1]) uses polygon patches clad in semitransparent 2D textures combined together to create a desired shape. This is also a rather simple and lightweight method, but it works best for stylized hairstyles rather than fur, which is naturally more uniformly organized (see figure 2.6).

Figure 2.4: Example of utilizing semitransparent 2D textures to visualize grass from Nintendo's The Legend of Zelda: Wind Waker (2003).

Figure 2.5: Example of utilizing semitransparent 2D textures to visualize grass from Animal Crossing: New Horizons (2020), and Ubisoft's Far Cry 6 (2021) respectively.

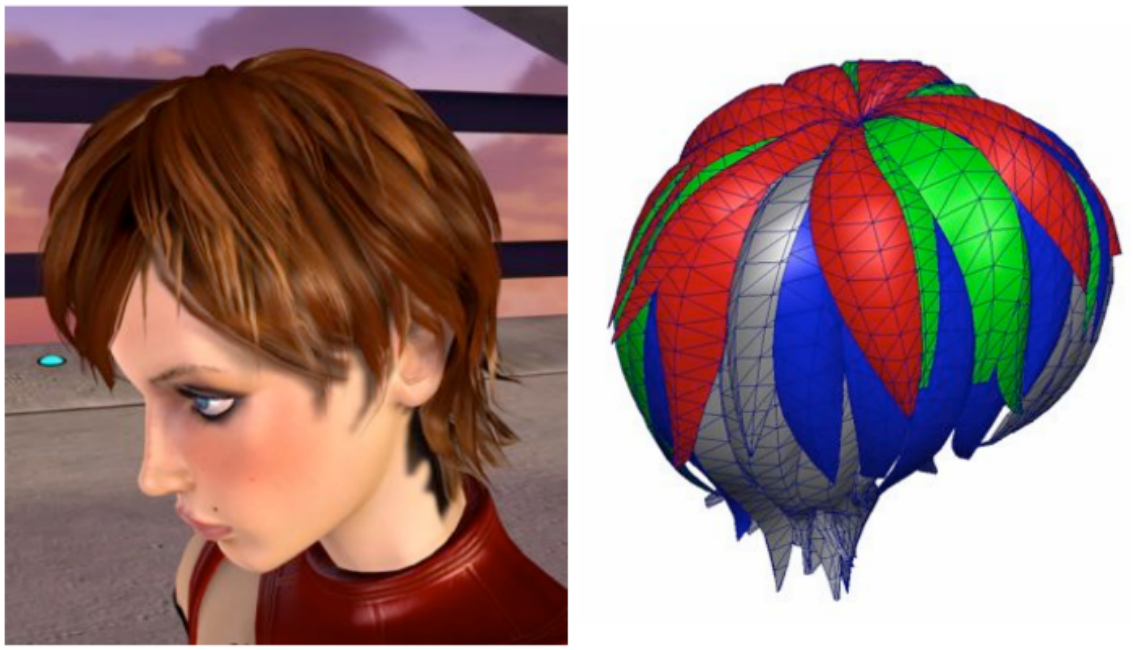

Figure 2.6: Example of using polygon patches clad in semitransparent 2D textures to create stylized hair[1].

In [21] the authors write about the benefits of using a Level of Detail (LOD) system when creating methods to render grass in real time applications. In this case three levels of detail are used. The first one is fully geometrically modeled grass straws (where the player is presumably very close to the grass). The second level uses a number of semitransparent 2D textured positioned vertically, plus a horizontal one covering the ground above the ground texture. The last, and least detailed level, uses only the horizontal texture from the previous level of detail (here the player is assumed to be really far away, and not able to make out individual details of the grass). Some variation of this is commonly implemented in most modern video games today. With sometimes additional levels added, or some left out, such as the very detailed geometrical level.

2.4.2 Shell texturing

Taking advantage of perspective, and the fact that grass and hair are fairly uniform elongated shapes, there are ways in which they can be rendered in very high-quality, with minimal processing power. Albeit with some constraints in regards to viewing angle, type of grass/hair, and viewing distance.

In [22] it is suggested that a lightweight method to render fur in real-time with convincing results using just semitransparent horizontal textures. They use something that they call shell textures, which is basically several shells of texture surrounding the mesh that you want to be clad in fur. All of the textures consist of multiple dots, each representing singular hair strands, with surrounding transparency separating

them. The texture can be thought of as a visualization of how a cross-section of the hair would look like (see figure 2.7).

Figure 2.7: Example of how a cross section of fur (shell texture) can look like.

Since all of the dots in the textures are aligned as the shells are stacked on top of one another, when you view the mesh and its' shells at an angle, the dots will form lines. These lines will appear three-dimensional. This is because parallax between the dots come into play, as opposed to when you are just looking at a static 2D texture on its own. The illusion dissolves however if the player looks at the shells too directly from the side; the angle at which the player can view the grass is decided by the distance between the individual shells, as well as the number of shells themselves. This problem is addressed in the paper, and resolved by using semitransparent vertical textures at the edges of the mesh. This does add a few more computations, but can be worth it to alleviate see-through in some scenes.

The power of shell texturing as a method lies in the fact that you get a result similar to what can only otherwise be achieved by geometrically represented individual grass straws, with no geometrical representation other than the individual shells. A famous example of the method being employed is in Nintendo's *Super Mario Galaxy*, where, most notably, a giant bee is clad in fur (see figure 2.8).

Figure 2.8: Queen Bee's body (from Nintendo's Super Mario Galaxy) is an example of shell texturing being used.

2.5 Decorations

By decorations we mean all "objects" added to the world, such as trees and rocks. For the placement of these decorations one technique has been particularly useful. This technique is called Poisson Disk Sampling [23]. This method is used not only for decoration placement but also for determining where dirt patches should be placed, but more on that in the method section.

2.5.1 Poisson Disk Sampling

The goal of Poisson disk sampling is to sample "random" points that are evenly distributed in the sampling space. This is great for the placement of trees and other decorations in our project, as the naive solution of simply spawning objects completely "randomly" results in a far less evenly distributed placements than one might

expect. See figure 2.9, for a comparison between completely random sampling and Poisson sampling.

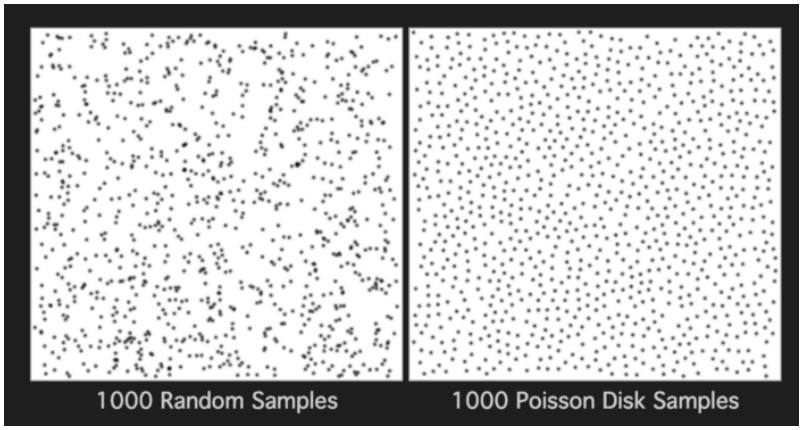

Figure 2.9: The difference in pure randomness (left) and Poisson Disk Sampling (right) [2].

The basic principle is, for two dimensions, divide your space into a grid and sample at most one sample per grid square. The grid ensures the even distribution while still retaining an element of randomness, because each sample is selected at random inside the grid squares.

The size of each grid square is determined by the predefined minimum distance r, the actual size being r / $\sqrt{2}$. The sampling then works by selecting an initial random point in the entire sampling space, inserting it into the grid, and marking it as active. The next step is to continue iterating for each active sample until there are none left. For each active sample, we generate up to k points (k is predefined) that are uniformly distributed in the spherical annulus between radius r and *2r*. For each of these points we test if they are sufficiently far away from any other existing samples, which if true they become sample points themselves and are marked as active. If k points are generated and no new sample is found, then the sample which spawned the points is removed from the active list.

2.6 Noise Functions

Something that has been utilized a lot for different graphical effects is noise. Noise functions are deterministic functions that return a value often mapped between 0 and 1. These functions can be constructed for any dimension, but in our case we have only used 2D functions. A 2D noise function takes in a 2D coordinate (X, Y) and returns a floating point value. These functions can be visualized as a 2D texture where each pixel shows the output for the function given the pixels coordinates, see figure 2.10.

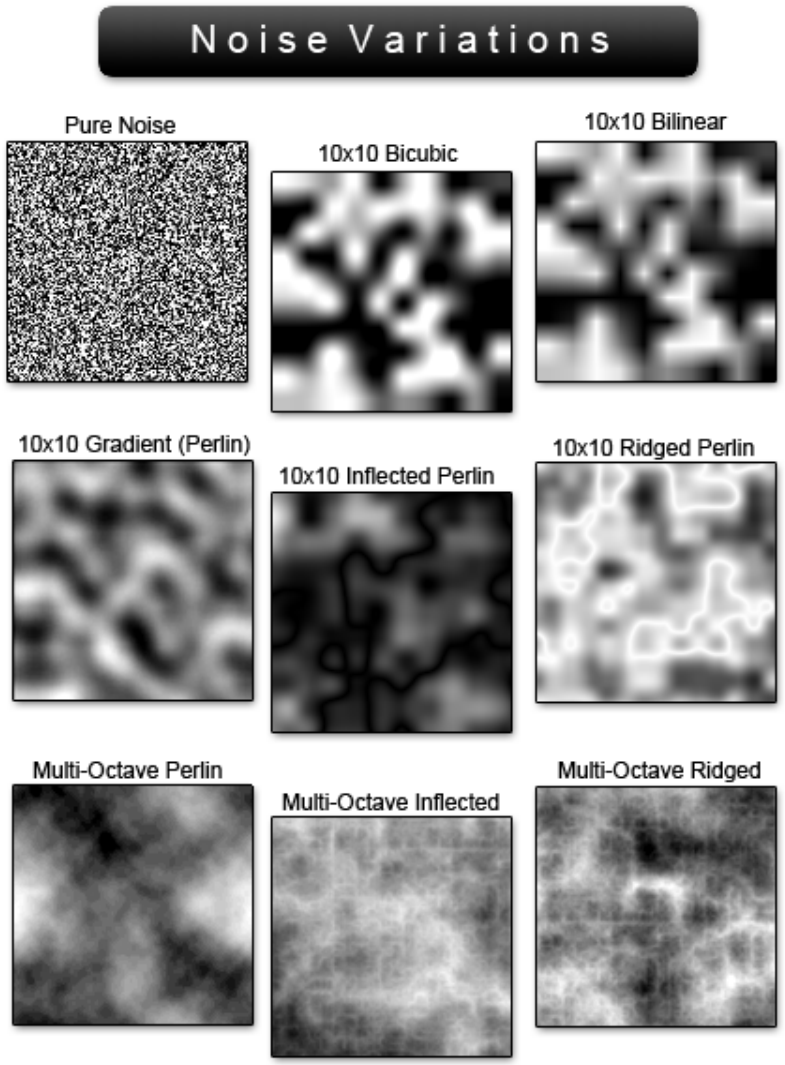

Figure 2.10: A number of different 2D noise functions displayed as textures [3].

There are many different noise functions. From more trivial functions, such as white noise which uses the input as a seed and returns evenly distributed ”randomness”, to more advanced, such as perlin noise[24] or simplex noise[5]. The use cases for these are endless, for example a terrain might be based off of a combination of noise functions to create a height-map as mentioned in section 2.3.

3
Methods

3.1 Generation Overview

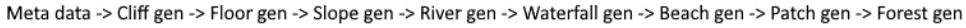

Figure 3.1: The generation is split up into multiple steps as outlined above.

The entire generation can be split into multiple smaller steps. These steps and the order in which they are computed is outlined in figure 3.1. The following sections will go through and explain each of these steps in detail.

The order of the steps are vital to the function of the algorithm, as is described in the detailed sections. The first step generates meta data needed by the following steps in order to decide how they should generate. The cliff generation uses the meta data to create the cliffs, forming the staircase-like terrain of Animal Crossing. When the cliffs have been generated, an algorithm goes through the terrain and fills in the floor of each elevation level. The terrain is divided up into tiles and each tile needs to have a mesh corresponding to the tiles terrain features. When the staircase terrain is formed, slope generation starts. Slope generation finds suitable locations where cliff walls can be replaced by a slope mesh of size specified by the user. River generation marks tiles as part of rivers an prepare connecting waterfalls for cliff transition. Waterfall generation then deals with all the special-cases associated with constructing a waterfall which follows the cliff formation. Once this is done the beach is generated, which makes sure to combine grid-meshes for both the ocean and sand planes in order to create a wavy coastline. Once the terrain is complete, it is populated by dirt patches and finally trees and rocks.

3.2 Cliffs and Slopes

The first feature focused on in order to create the Animal Crossing world generator was the cliff generation. The cliffs, as described in section 1.3, is key to capturing the general shape of what makes an Animal Crossing style world. When looking at traditional methods of terrain generation, see section 2.3, it became clear that a

different approach would be needed.

Heightmaps are not suitable by themselves due to the inherent flaw of not being able to create vertical cliffs in the first place. Using heightmaps to construct the cliffs will always be constrained in their vertical angle by the resolution of the grid.

Using a voxel based approach was considered as it had the benefits of being able to represent both vertical cliffs and the planes in between. However, having an entire voxel based system for the sole purpose of representing vertical cliffs that only cover a minor portion of the terrain is seemingly unnecessary. The real benefit of using a voxel based terrain representation is the modifiability it affords, which is the key aspect we investigated with a voxel based approach.

The cliffs need to adhere to a specific structure in order to capture the Animal Crossing style. They need to both follow the borders of acres bordering different elevation levels, while simultaneously traversing in a non-linear way. An idea that spawned from this was to create a voxel terrain where each acre is set to their respective elevation levels, creating vertical cliffs outlining the borders as described, and then modifying the voxel data by ”eating away” at the cliffs, giving the structure desired. This approach would also allow handmade modifications to take place post generation, either from a designer or through game interaction akin to that in New Horizons.

The issue faced here is that such a voxel based system inherently requires any voxel to be modifiable and still allow the ”puzzle” to fit. This then leads to the shapes used to fit everything together being more homogeneous, leading to the same design traits of New Horizons, which is undesirable. There must be a different solution.

The solution settled on is a hybrid approach taking elements from a number of different techniques. The terrain is built in a top-down fashion, which means that we settle on the macroscopic features before doing the detail work. It should be noted that the descriptions to follow may sometimes refer to planes and directions in the form of X, Y, Z directions. X and Z together create the floor plane while Y is used as the up direction. To clarify, this means that a change in elevation is equivalent to a change in the Y direction.

3.2.1 Cliff Solution

We will start by describing the solution for the cliff generation in more high-level detail in order to get a sense of what is happening. Following sections will fill in with more concrete details.

The generation starts by determining the size of the world and then computing meta data for each acre. The first piece of data determined is the elevation of an acre. A tile within an acre may not be of this elevation at the end of world generation, but

it is used as a starting point to determine what elevation level a cliff or floor tile within the acre should have.

Once all acres have had their elevation determined, islands are calculated. Islands refers to acres that are connected on the same elevation level. This is useful meta-data to know how acres are encapsulated, which is useful when determining which acres will have a continuous cliff between them or where slopes should be placed.

Another piece of meta-data that is computed is cliff orientations. This refers to acre-level data that helps determine what cliffs an acre will contain and which directions cliffs should turn during cliff construction in that acre.

When this acre-level meta data has been computed, the cliff construction can begin. Cliffs are constructed from highest elevation to lowest elevation, this order is important as lower elevation cliffs can be dependent on higher level ones. Cliffs are constructed on a per island basis. This means that for each island, which by definition should have a cliff, we find the acre where cliff construction should begin and then create a cliff-walking agent to begin cliff construction. The cliff-walking agent then iterates through its cliff construction procedure using a combination of modified techniques discussed in the background section and custom constraints defined to adhere to the terrain style. During cliff construction there are a number of special cases that must be dealt with in order to keep the terrain mesh solid.

After cliff construction, the final elevation for each floor tile is determined. First the standalone floor tiles are determined, and then the floor level of each cliff tile is determined in a separate pass. Once this is done, slope placement is computed and executed on.

Finally, when all data is generated to determine formation of the terrain, the actual mesh creation takes place. This is done by merging multiple smaller tile meshes with each other to create a larger terrain mesh.

3.2.2 Acre Meta Generation

Firstly, the size of the world is set. This is done by the designer before generation by detailing the number of acres in the X and Z direction, and the size of each acre in number of tiles. The acres are then represented as a matrix (two dimensional array), with each entry containing the data for the corresponding acre. This structure is intuitive for its eventual geometric representation. In our demo, the first acre: acre (0, 0), is located at furthest Z coordinate in world space, meaning an acre with higher Z position will be closer to the 0 coordinate in world space Z direction. X is mapped normally.

3.2.2.1 Acre Elevation

The idea now is to, taking inspiration from heightmaps, covert the acre matrix into a heightmap of its own. Each acre is assigned an elevation level, which will form the basis of the terrains layout, directly inducing the placement and orientation of cliffs and floor tile elevation. This assignment is done through the following procedure.

First the two northern most rows of acres are set to the highest possible elevation. Then the southern most row of acres is set to the lowest elevation. Next we iterate through each column of acres from north to south, not including the already set acres. For each acre, we look at the previous acre to the north, take its elevation, and then there is a 60% chance that we make this acre 1 elevation lower, else we stay the same elevation.

3.2.2.2 Islands

Once the base elevation of each acre is set, the islands are determined. We define an island as a group of acres where each acre has the same elevation and is connected to all other acres of the island. Connected, in this context, refers to acres which you can walk between by traversing acres Either in the X direction or Z direction each step. These islands are useful for later steps.

3.2.2.3 Cliff Orientations

Another piece of data that must be determined at this point is what will be referred to as cliff orientations. Cliff orientations refers to meta data stored for each acre detailing what edges and corners of that acre will be traversed by cliffs. This will become relevant when the cliff construction procedure is outlined later.

There are three edges that a cliff can traverse along, the western, southern, and eastern edge. The northern edge of an acre will never have a cliff wall due to the acre elevation procedure defined. A cliff can also traverse an acre without covering an entire edge, instead there are situations where a cliff only needs to traverse a corner section of the acre. In total there are five possible orientations that need to be accounted for, the three edges mentioned and the south-east and south-west corner traversals. South-west corners only need to be traversed when there is a southern cliff orientation leading into the acre from the west and a western-edge orientation in the next acre to the south when leading out. Northern corners don't need to be taken into account because there is no northern edge possible. This description is easiest understood when examining figure 3.2.

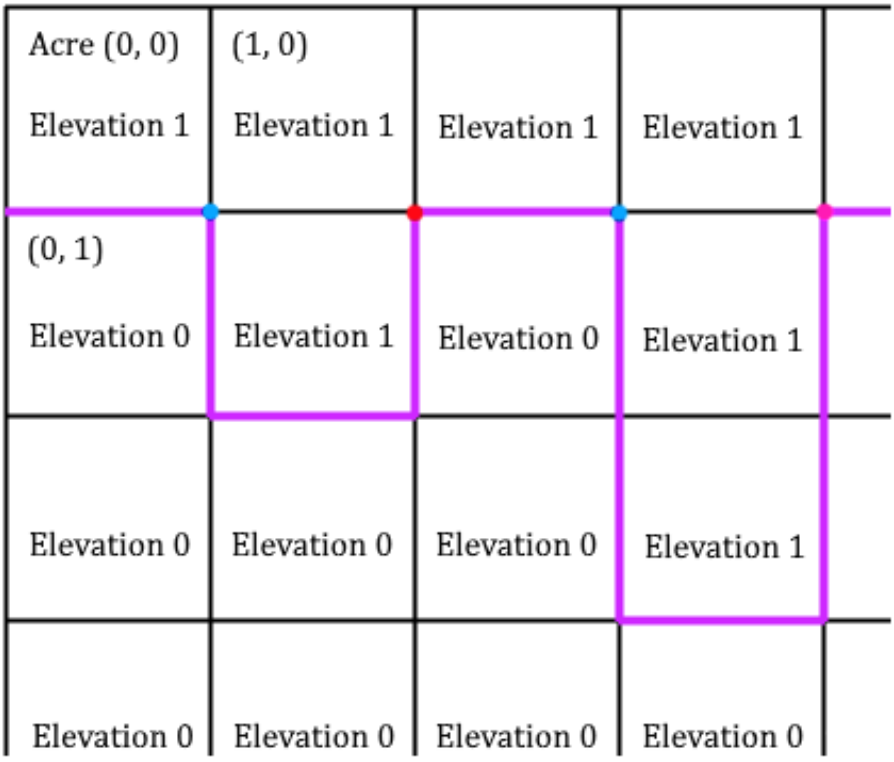

Figure 3.2: The purple lines show the west, south, and east cliff edge orientations. Acre (0, 0) has the south cliff orientation marked, acre (1, 1) has all three, west, south, and east marked, and so on. The blue dots represent marked south-west corner cliff orientations, and the red dots south-east. Important to highlight is that the first blue dot is part of acre (1, 0), not acres (0, 0) or (1, 1).

An acre determines its cliff orientations by comparing its elevation level to those in neighbouring acres. To determine if an edge will have cliff traversal, the acres elevation is compared to the neighbouring acre in the direction of that cliff. If the neighbouring acre has a lower elevation, then that edge is marked as a cliff orientation of the acre. Corners are determined by comparing the three surrounding neighbour acre's elevations. If the diagonal neighbour is lower than the two other neighbours, and the two neighbours are of the same elevation level or higher than the current acre's elevation. Then, the corner cliff orientation is marked as part of the acre. The reason we check for not just the same elevation level, but also if its higher, is because of a special case that may occur.

Two separate islands may in some cases share a single continuous cliff, more on this later. This happens when the two islands, of the same elevation, share a corner, but the acre between them is of a higher elevation. In order to achieve an interesting terrain feature, a ledge between two islands, the higher elevation acre is marked with the relevant corner cliff orientation. By doing this, during cliff construction a cliff that normally would wander into the higher elevation acre until it hits another cliff,

will instead turn when entering and sometime miss the higher elevation acre cliff. Instead, it will continue into the other island of the same elevation and continue by constructing its cliff as well. This will then lead to a ledge inside the higher elevation acre connecting the two islands. This explanation will be more clear once the rest of the cliff sections have been read for context.

3.2.3 Cliff Construction

Once the acre meta data has been generated it is time for cliff construction. Cliff construction occurs by spawning what we define as a cliff walking agent. The agent performs an iterative depth-first walk based on L-Systems with constraints defined in such a way as to create the desired cliff formations.

As the entire world is tile-based, we define rules akin to those in L-Systems in terms of what cliff tiles may be generated given the current state. Our alphabet consists of the different cliff tiles, and our production rules define which tiles may follow a certain tile. We deviate from traditional L-Systems in that we may revert an iteration if certain constraints are breached. If a step is reverted, then that possible branch (of the branches of possible steps to take) is discarded and a different branch is tried. If no branches are possible then we can revert further steps back, thus giving us a depth-first search to the problem space.

3.2.3.1 The Tile Set

For our cliff construction to work, we must define a tile set (the alphabet). In order to fulfill the goals stated for the cliffs formation, the tile set must be selected in a manner that both ensures a solid and completable mesh when strung together and a free-form pattern to its formation. For example, a simple tile set would be to include 3 straights (west, south, and east) and 4 45° diagonals. This would ensure a solid and completable mesh, however this would result in the same cliff formation as that seen in Animal Crossing: New Horizons, which is not desirable.

The tile set that was ultimately settled on consists of 26 tiles, which can be reduced down to 4 unique tiles when discounting rotated, mirrored and inverted duplicates. These cliff tiles were selected by imagining a tile as divided into 8 nodes and selecting edges based on possible combinations, excluding the diagonal resulting in a 45° angle, see figure 3.3. Through rotation, inversion, and mirroring we end up with the final tile set used, see figure 3.4.

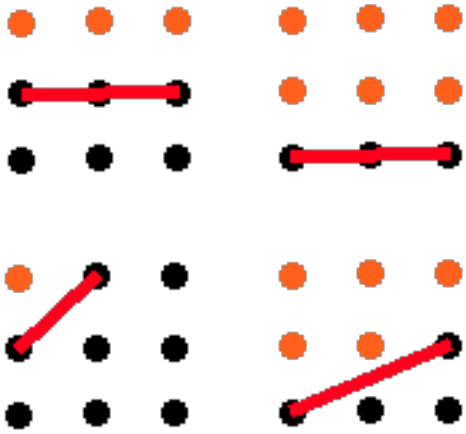

Figure 3.3: The 4 unique cliff tiles when not considering rotations, mirrors, or inversions as unique. The Lines indicate the actual wall mesh and the orange dots represent being inside the cliff while the black dots are outside.

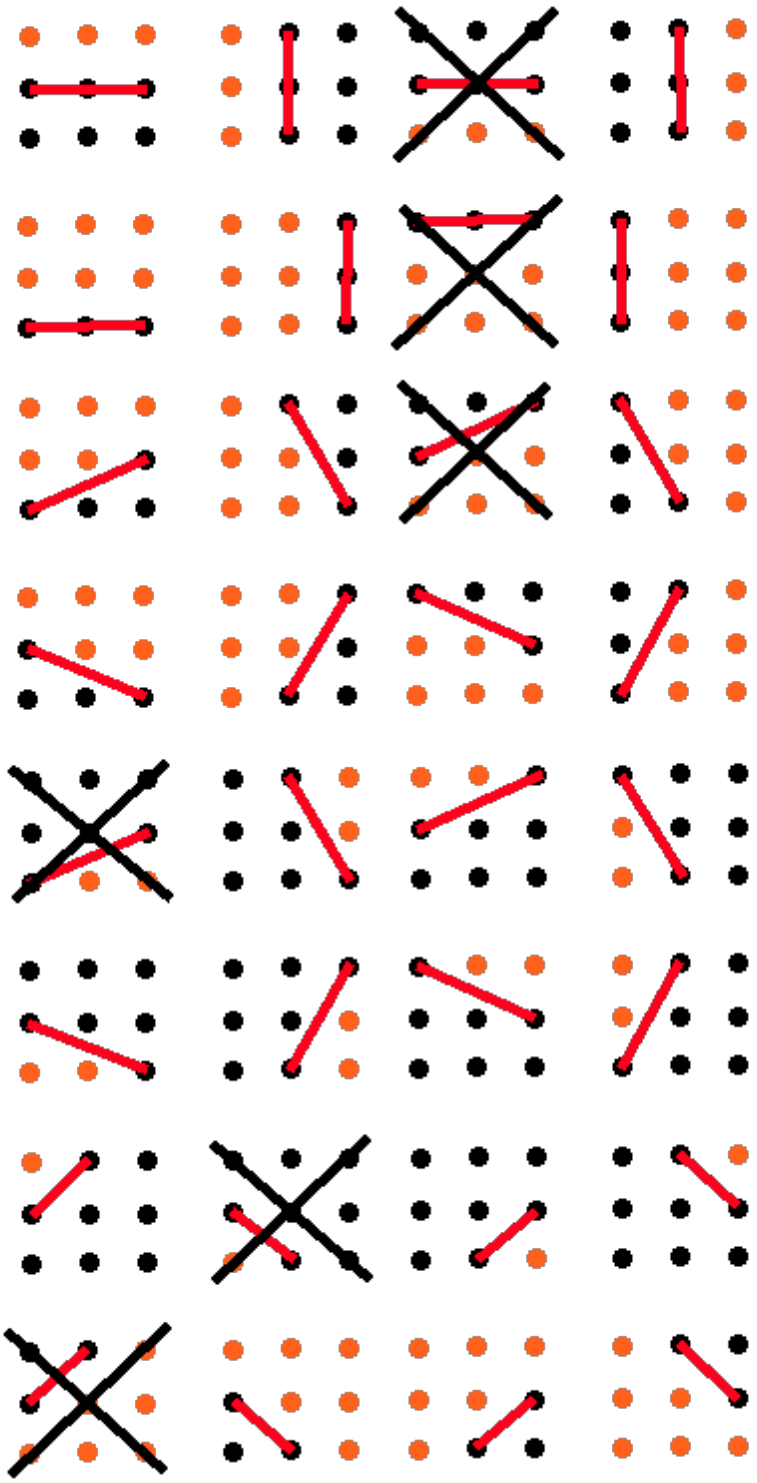

Figure 3.4: The 26 different cliff tile orientations used. The Lines indicate the actual wall mesh and the orange dots represent being inside the cliff while the black dots are outside. Possible variations that are not used are crossed out.

With our alphabet in place we can define the production rules. Each cliff tile has a list of cliff tiles that may follow it based on the current state. The state refers to all steps taken up until the current iteration, however in practical terms what matters when selecting the next production rule to apply is the current cliff tile and the current prevailing direction. Because of this and the graphical nature of the project, production rules are defined for each cliff tile in terms of a direction, an offset and a cliff tile. The direction indicates if the rule is applicable for the current prevailing direction (more on this later) and the cliff tile specifies the next cliff tile. The offset is defined in order to know the relative placement offset of the next cliff tile in order to figure out its position in the world when constructing the graphical mesh.

There is more data needed for each tile in order to decide on which production rule to use. The first is whether or not a tile is an end tile. An end tile is a tile that if placed at the eastern edge of the world, will connect to the edge of the world. The second piece of data stored for each cliff tile is which other cliff tiles it overlaps with, this is important when separate cliffs should merge together. Two cliff tiles overlap when their edges intersect and the edges face outwards in the same direction. Without knowing this information, holes will be created in the terrain mesh.

Calculating and filling in all this information manually for 26 tiles turned out to be too big of a task and prone to errors, therefore we developed a separate program to calculate the values for us.

3.2.3.2 Cliff Walk Agent

The acre meta data has been computed and our tile set is defined, including the production rules, now it is time to do the actual cliff data generation. The world consists of tiles, where each tile also belongs to a uniformly sized acre. An acre has a meta elevation assigned to it, meaning that if two acres lie next to each other with different meta elevations then there must be a cliff between them. All acres with the same elevation and that are connected by at least one edge are considered part of the same island. This means that there should be a continuous cliff for each existing island and because a cliff is always constructed within the higher elevation acre unable to cross over into the lower elevation one, this means that the higher elevation acre will have the possibility of containing tiles that are of both the acre's elevation and lower. Important to note is that an acre may be both the higher elevation acre in one instance, and the lower elevation one in the next.

For each island, sorted in highest elevation to lowest (this is important), we start by first finding the acre from which to start cliff construction. This is done by checking each acre top to bottom from left to right and finding the first occurrence of the current island we are working on. If the acre found has a western or southern cliff, we are done, otherwise continue searching until these requirements are met. Remember that this information has been precomputed as per the cliff orientation section. We do not check for an eastern cliff in a starting acre, since a cliff can never start in that orientation as per the definition of our terrain.

Once the starting acre is decided we must also find the starting tile, in order to do this we must first do some additional checks. Because cliffs split the elevation levels of tiles in the higher elevation acre, this means that a starting acre cliff is not guaranteed to be connected to the neighbouring higher elevation cliff, creating holes in the mesh. For example, if a starting acre is to start with a southern cliff and the neighbouring acre to its west contains an eastern cliff, then the two cliffs may not be physically connected as seen in figure 3.5. For this reason it is important that the cliffs are constructed from highest elevation to lowest elevation, as will become clear.

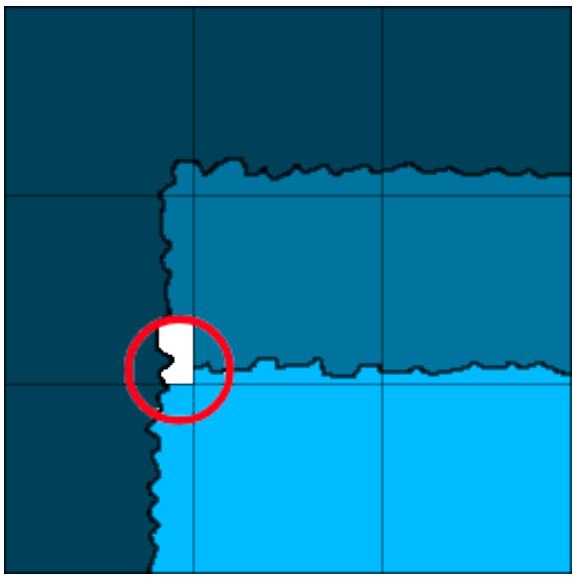

Figure 3.5: The cliffs of different elevations don't necessarily sit together if a cliff starts construction from only its own first island acre. The squares represent acres and the colors indicate different elevations.

To solve this issue, if the starting acre has a western cliff we check the northern neighbouring acre. If the neighbouring acre has a southern cliff, we must then find the starting tile within that acre instead of the starting acre. During the cliff walk procedure, a prevailing direction must be set in which the agent is traveling. We set the initial prevailing direction together with the starting tile, which in this first case would then be south. If the neighbouring acre instead has a south-west corner cliff orientation and its western neighbour in turn has an eastern cliff (which should always be the case), then we find the starting tile within that acre instead of the starting acre and set the prevailing direction to be east. If the starting acre does not have a western cliff, but instead a southern one, then we check if the neighbouring acre to the west has an eastern cliff. If this is the case, then we find the starting tile in that acre instead of the starting acre. See figure 3.6. When none of the cases above hold true, then the starting tile is selected from inside the starting acre.

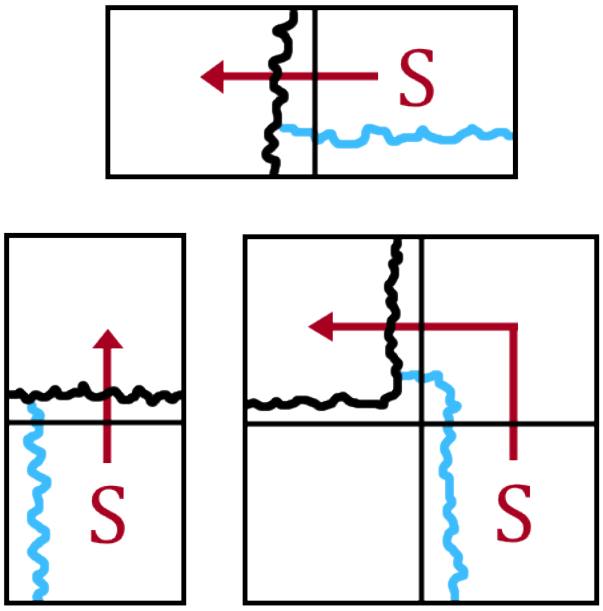

Figure 3.6: The three possibilities when having a start tile in another acre than the starting one. Each square indicates an acre, the S indicated the starting acre, and the arrows indicate the path traversed to find the acre where the starting tile needs to be.

When the starting tile's acre is known, then the actual starting tile must also be selected. When selecting the starting tile, maximum and minimum cliff eat constraints must be taken into account. The cliff eat constraints are constraints predefined that set the limits of how far in and out a cliff may be built inside an acre. The maximum cliff eat defines how far in from the acre edge a cliff may wander, while the minimum cliff eat defines how far in a cliff must be at a minimum. For example, when constructing a western cliff of an acre, a cliff tile may only be placed at least cliff eat minimum from the western acre edge and at most cliff eat maximum.

In order to find the starting tile, if inside the starting acre, we find the acre edge where the cliff should start trivially by prioritizing in the order: western, southern, eastern (in practice the first edge can not be an eastern edge). Once the edge is known, we select a random value between the minimum and maximum cliff eat edge constraints to use as offset from the acre edge and start at the beginning of the edge in the prevailing direction. In concrete terms, if the southern acre edge is the starting edge, then starting tile would be the western most tile offset a random value north from the southern edge.

In the case where the starting tile is not part of the starting acre, a different approach is required. In these cases we iterate from the edge shared with the previous acre against the prevailing direction, checking all possible values between the minimum and maximum cliff eat values along the way. These values are checked from the edge inwards. We stop checking once a cliff tile has been encountered, ensuring

that the terrain is stitched together, this then becomes the starting tile.

Once the starting tile is known, a cliff walk agent can be created to perform cliff construction. It is the cliff walk agent's job to perform the L-System aspect of the terrain generation. A cliff walk agent's task is to start at the starting tile, then iteratively perform steps in the form of applying (or reverting) production rules which place new cliff tiles and update the current state. When placing a cliff tile, the cliff tile is placed adjacent to the current position by the offset specified in the production rule applied. This then also updates the position of the agent, so that the next step will continue constructing a continuous cliff.

A cliff agent will walk according to the prevailing direction, set according to the cliff orientation when spawned. When walking, cliff orientations are taken into consideration in order to direct the agent along the correct path. The position of a cliff walk agent and the current acre's cliff orientations determine if there should be a change in the prevailing direction. For example, while walking down the western edge and the acre also includes a southern edge, then at some point the prevailing direction will be changed from south to west. The point when this is changed is determined by the minimum and maximum cliff eat constraints, as the only logical place where a change could happen is at tiles which are valid placement tiles for both edges. Direction changes occur as you would expect intuitively based on the cliff orientations, so when entering a new acre a south-west corner orienation for example will immediately shift the prevailing direction from east to south.

Each step taken by an agent will always move one tile in the prevailing direction. When an agent is traversing an area where a direction change should happen, there is a uniformly distributed chance of a direction change happening along the prevailing direction axis.

Given this system of an agent walking along constructing a cliff by applying the production rules defined, and taking into account min/max constraints when selecting which rule to apply, a squiggly cliff line will start to form along the edges of islands. The walk agent considers itself done when either colliding with another cliff, or reaching the end of the world.

When spawning cliff walk agents for each island in order of elevation, Animal Crossing style cliffs start to emerge. However, all is not well. There are a number of special cases that must be dealt with and even greater, the possibility of reaching a dead end where no production rules are applicable. In order to deal with this last possibility, each agent keeps a complete history of all steps applied. When a dead end state occurs, then instead of applying a step the agent reverts a step according to its history data. This means that all state changes that can occur during an apply step must also be revertable. An agent will therefore work in a depth-first fashion when finding a suitable cliff solution. It is a given then, that an agent must also keep track that the same branch is not explored multiple times.

Special cases must also be handled. One of these is tiles where two or more cliffs merge. In these tiles, which we call merge tiles, care must be taken when selecting what cliff tiles can be used. If cliff tiles are selected as normal, then holes in the terrain mesh may occur as seen in figure 3.7. Therefore, as mentioned earlier, each cliff tile also contains information of which cliff tiles it overlaps with in such a way to create a seamless mesh. When selecting a production rule to apply which will end up as a merge tile, then we also check that the cliff tile that will be placed is valid relative to the other cliff tiles in the merge tile square.

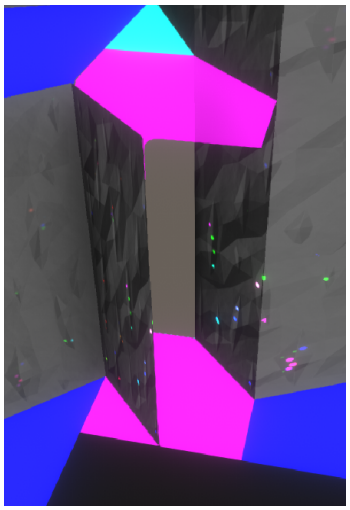

Figure 3.7: A tile containing two cliff tiles. In this case, proper care was not taken to see that the cliff tiles were compatible with each other in order to create a cliff without holes.

Another special case that must be handled has to do with when an agent is done. As mentioned, an agent considers itself done when colliding with another cliff or reaching the edge of the world. However, due to the tiled nature of its movement, it is sometimes possible for an agent to walk through an existing cliff without noticing. This happens when the two cliffs do a diagonal walk at their merge point, see figure 3.8. To solve this issue, for each step an agent takes it both checks if it has reached another cliff tile, or if it has passed through a cliff by checking the relevant neighbouring tiles.

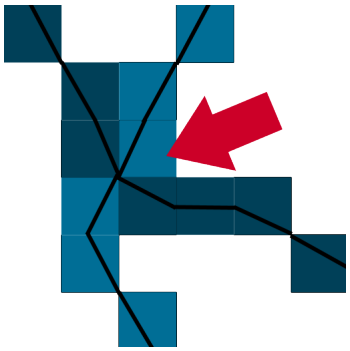

Figure 3.8: Two separately walked cliffs, indicated by the difference in blue hue. If collision is only checked for the actual tile you apply a cliff tile to, then two cliffs may walk right through each other.

The final special case that must be dealt with requires some additional knowledge on how the cliff walking works. When an agent is spawned for an island and is set free to do its walk, it won't always end after the last acre of the island. Due to the nature of how the cliffs are formed, eating into an acre, all cliff collisions always end inside a neighbouring acre with higher elevation and not part of the island the agent was spawned in. Or, if the island is separated from another island of the same elevation by a single corner, as in figure 3.9, then south-west or south-east cliff orientations will have been generated for the acre of a higher elevation, for the use for the lower elevation acres in order to allow for a seamless connection between them. Meaning, an agent spawned for one island, may also complete the cliff construction for another island of the same elevation if the walk agent does not collide with a cliff while being inside the acre of higher elevation. Allowing this behaviour results in a unique terrain feature we call corridors which will be highlighted in the results section.

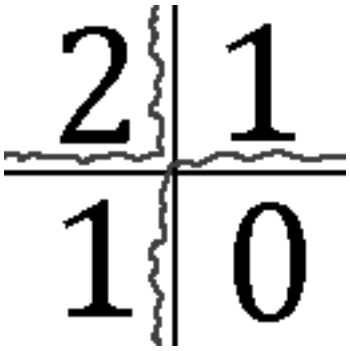

Figure 3.9: 4 acres that share a corner. The numbers represent the elevation of each acre. In this case, the two acres of elevation 1 would be part of different islands. However, a single cliff may still be used to construct both of the islands cliffs as the higher elevation acre will have a south-east corner orientation marked on it for this purpose.

The special case that must be handled relevant to this, is when an agent does not successfully pass through the acre of higher elevation and instead collides with a

cliff. When an agent is spawned for the next island, it will start searching for a starting tile inside the same acre where the previous agent ended. The starting point is found by searching for a cliff tile, which in this case may belong to the cliff from the island of the same elevation created by the previous agent. If the current agent would just naively start building the next cliff section from here, then it runs the risk of creating an internal cliff as seen in figure 3.10.

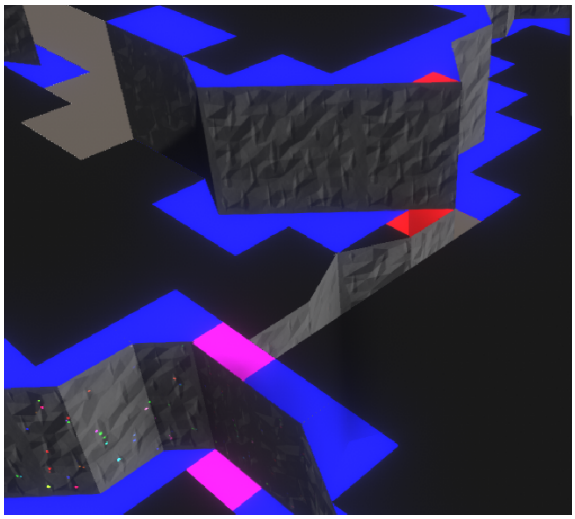

Figure 3.10: An unhandled internal cliff can be seen in the middle of the image.

In order to deal with this an agent must, when starting from a cliff of the same elevation, find all cliff tiles that will end trapped inside and resetting them as being normal floor tiles. It is easy to know which tiles will end up trapped, as it is simply all tiles after the starting tile in order of construction. In order to find these, each tile that is marked as a cliff in the world keeps a reference to both tiles it is connected to, the previous and next. These are set by the agent during construction, since it knows which tile was created before and which tile it creates after. In order to remove the internal cliffs, it is then as trivial as starting from the starting tile and simply looking up the next connected tile and removing it. Finally, when all internal cliff tiles have been dealt with, the new connected tile of the starting tile will be the first tile constructed by the new agent.

One caveat that should be noted is that of merge tiles; a tile which may have multiple cliff tiles in it. These must naturally keep track of more connections than just the two of a normal tile. This system of keeping references to the connected tiles of a cliff tile will come to use when computing floor tile elevation as well.

With all these cases dealt with, we know have the basic cliff structure for the Animal Crossing style terrain, as can be seen in figure 3.11. Note that the floor tiles here are displayed correctly, how this is computed is explained next.

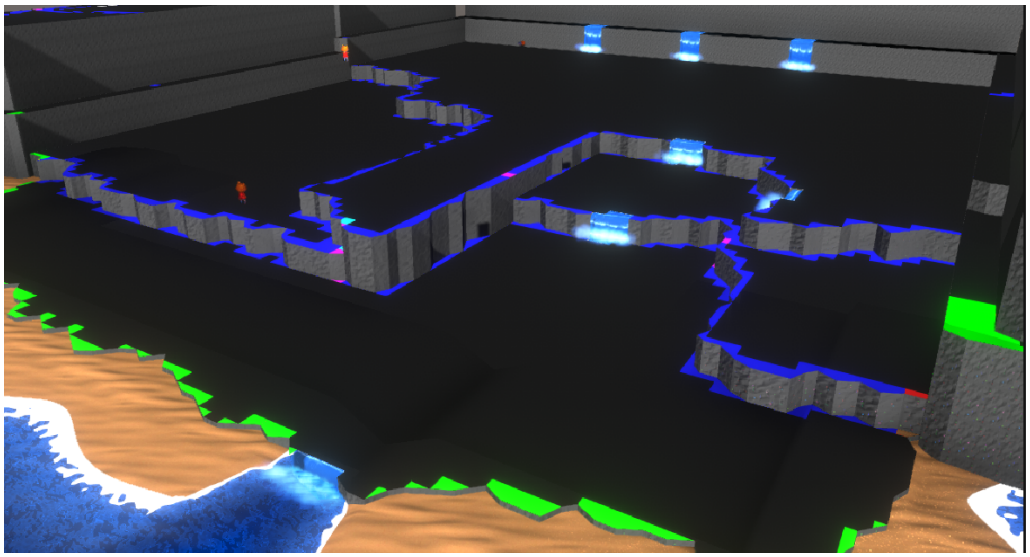

Figure 3.11: A view of the completed cliff construction for one world. Ignore the waterfalls, and surrounding terrain.

3.2.3.3 Floor Tile Elevation

The floor tile elevation problem has a number of steps to it. The first of which is to solve the elevation level of the purely floor tiles. By this I mean the tiles which only contain a flat floor and not a cliff tile. This is done by iterating over every floor tile on in the world and doing one of two things. Either, if its already been ”leveled”, then we skip it, otherwise we check each of the four neighbouring tiles that share a complete edge and ”link” them if they also are purely floor tiles. While linking we continuously update what the lowest elevation we have found is and mark the tiles as ”leveled”. Each floor tile is by default initialized to the acre elevation height. When no more tiles to link are found, we set all linked floor tiles to the lowest found elevation level. This ensures that at least all pure floor tiles have the correct elevation, when dealing with tiles with cliff tiles inside however, we must do a bit more work.

In order for there not to be holes in the terrain, tiles with cliffs in them must also include a floor mesh and a roof mesh. The roof meshes are trivial, their elevation is simply equal to that of the cliff. Each cliff tile has a custom roof mesh to go with it so that we achieve an edge following the cliff formation.

Cliff floor tiles are a little more work. The floor tiles do not need to be custom, since you won’t see the part inside the cliff anyway. Because we support cliffs being higher than a single elevation, determining the elevation level of the floor is not known trivially. During the previous normal floor tile evaluation, during the connection phase, if a cliff tile is found, then that tile is not added as connected, but it does save the final elevation level of the floor tile in the tile with the cliff. This means that cliff tiles can set their floor tiles according to this saved value. However, not all cliff tiles will have been found during the connection phase. Connections only occur between tiles that share an entire edge, this means that two tiles can not

be connected diagonally, therefore some cliff tiles will not border any pure floor tiles on its lower side, see figure 3.12.

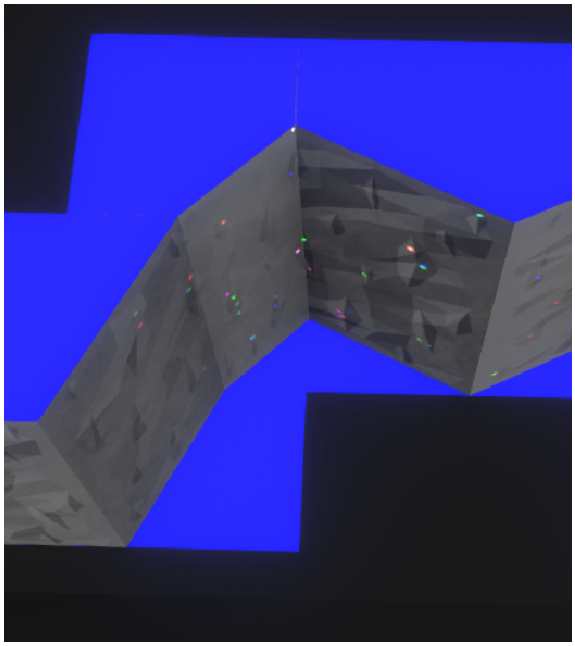

Figure 3.12: The upper left cliff tile does not share an edge with any lower elevation pure floor tiles.

To solve this, we once again utilize the data stored of what cliffs are connected. A cliff tile that does not have an immediate value to set its floor elevation to, will instead start searching its neighbours in both directions until one is found. It will either find a normal tile with an elevation number or reach a merge tile, where upon it will select the next floor elevation down stored in it.

3.2.4 Slopes

When the cliff generation is complete and the floor levels have been computed, it is time to find where slopes can be placed and generate the necessary data for their creation. Slopes are defined as by a width and height, taking up a rectangle of space in terms of tiles. A slope must of course have one end on a higher elevation than the other, with the constraint that the elevation difference must be no more than one.

One of the design goals was to create slopes which integrate into the cliff sides, and not just an external addition to the side of a cliff. Finding the placement of a slope is quite simple. First of all, each island spawns at most a single slope for each other island it can connect to. This ensures traversability to all sections of the world, while keeping the number of slopes low. If multiple acres in an island can contain a slope that connect to the same island, then one of the acres is chosen at random to contain the slope. Once an acre has been identified to contain a slope, we must

find the exact tile location for it. This is done by iterating through each tile of the acre with higher elevation, and in each step checking if the placement is valid. Valid placement is determined by checking that the tiles of the first and last row/column (depending on orientation), based on the width defined for the slope, are all pure floor tiles and of the correct elevation. Once all valid placements have been identified, the valid placement closest to the center of the higher elevation acre is selected.

Placement of the actual slope requires a few steps. First of all, tiles inside the slope area are unmarked as cliffs and instead marked as slopes, meaning we remove the cliffs where the slope is to be placed. For the sake of brevity we will use width to mean tiles across the slope not in the walking direction, and height to mean the opposite or row/column respectively. The next step is to mark the outer most tiles in the height direction special. In each of the two outer columns, we mark the first tile to be a custom end tile for the high end, then we mark each consequent tile as a slope cliff until we find the last tile that used to be a cliff. This tile is then marked to be a custom end tile for the low end. All of this ensures that the formation of the slope adapts to the cliff formation, making the slopes seem more natural.

The custom top end tile and slope cliffs are always the same, as they only need to take into account connecting smoothly to each other. However, the custom bottom end tile is more tricky. This tile is responsible for connecting the neighbouring cliff tile with the first slope cliff. In order to do this, each cliff tile has for each possible slope connection possibility, six of them (two for each orientation), an index indicating which cliff tile to use in order to seamlessly connect to the slope cliff. In our program, slope cliffs are all defined as straight pieces, meaning all connections are to go from the end node of the cliff tile to the relevant middle node dependent on orientation. This makes calculating the correct connection piece easier.

Unfortunately, there isn't an existing cliff tile piece for every needed connection. However, fortunately only a single special cliff tile is needed to solve this issue. See figure 3.13 for an example of when the special connection tile is needed.

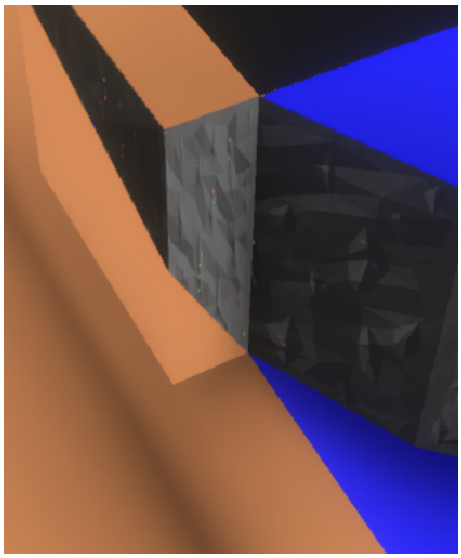

Figure 3.13: The special slope cliff tile. This sharp turn is sometimes required in order to properly connect up to the next cliff tile.

The last part of the slope construction is to number each row of the slope, so that during mesh construction the floor tiles in between the slope cliffs can be rotated and scaled properly in order to create seamless slope connecting the two elevations. The floor tiles that end up ”outside” the cliff, must be vertex manipulated a bit extra in order to line up on all sides of course. Also, two floor tiles need to be handled separately from the rest. These are the floor tile meshes that are used in the connection tile and require an extra vertex in order to connect up all the ends. See figure 3.14 for an example of a completed slope.

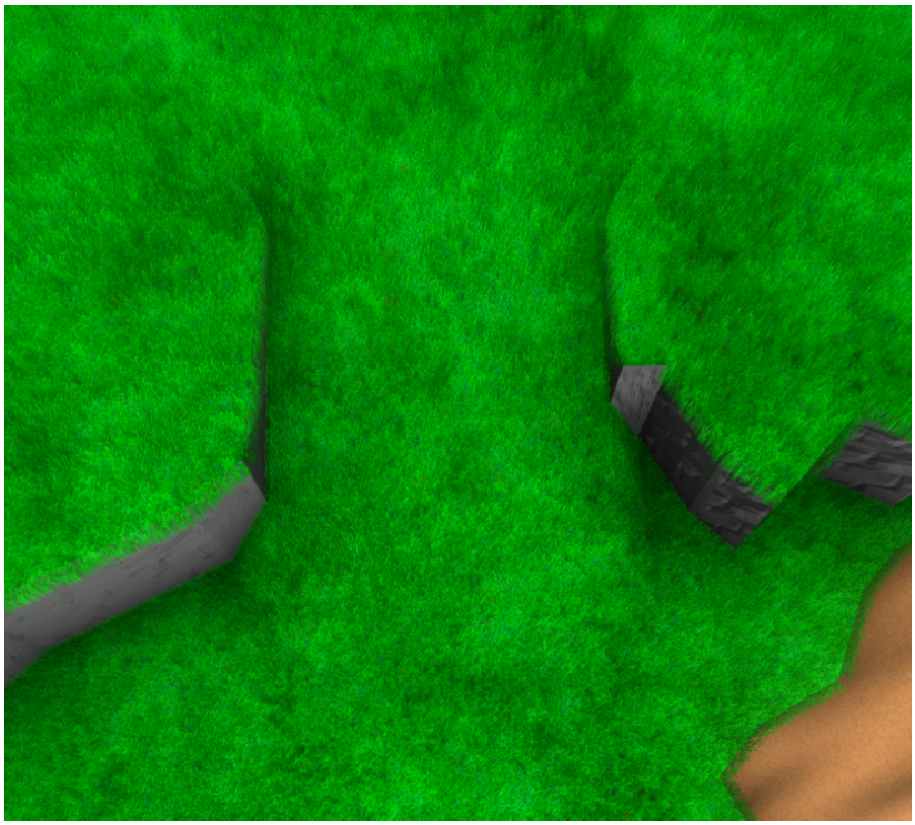

Figure 3.14: A completed slope in the world. As can be seen, the slope is integrated into the cliff, and the special slope cliff tile can be seen on the right side.

3.2.4.1 Mesh construction

Once the generation is complete in terms of the data for the terrain, it is time to create the actual mesh that will be rendered on screen. This is done by iterating through each tile and spawning relevant meshes dependent on the generated data. This includes the floor tile meshes and cliff tile meshes, but also other features part of the generation not yet discussed such as trees, rocks, rivers, and waterfalls.

When meshes have been spawned for each tile and mesh data manipulation has been completed, it is time to combine all these sub-meshes into one big world mesh. This is done to reduce draw calls. All generated features however, can not be joined into the same mesh as they require different materials with different shader programs. In our program, the meshes were combined into the following: Flats, Cliffs, Rivers, RiversBottom, BeachCliffs, Beach, Ocean, Canopy, Rocks, Trunk, and Waterfall. To reiterate, this is done so that each of these elements can use a different shader program when rendering while also reducing the number of draw calls from a few thousand to ten.

For our demo, where you can free fly and see the entire map all at once, this single world mesh works well. In a game however, which is the main application we imagine this to be used for, the most logical would be to break the mesh into a per acre basis. Seeing as the Animal Crossing style camera view only ever allows the

player to see a limited space, this would free up resources that the computer uses to calculate data for most of the mesh that ends up being clipped off camera.

3.3 Fur and grass

We wanted the grass to have a visually pleasing and realistic look to it. A lot of grass was going to be rendered in our scene, as such we were aware that there was the possibility to run into serious performance issues, depending on how we chose to implement the it. We had to look for ways to achieve that without geometrically modelling individual grass straws on the ground. We utilize the techniques presented by [22], which is designed for creating realistic fur in real time. We decided on this method because it uses optical illusions in order to get grass (or fur) where you can make out individual straws, without representing them individually with 3D meshes. We use this technique also because the technique works particularly well in our scene, that is a scene where the camera rotation and distance remains relatively static, as compared to a game where the player has full control of the camera or a first person shooter, where the illusion would break.

3.3.1 Fur shells

Our first naive solution to the problem was to try out the method by manually making semi transparent layers of our character in the game. This gave us a sense of how well the technique itself would work for the problem at hand. In order to find the optimal settings and tweaks for the parameters, the subsequent step was to write a shader that we could customize in real-time. In order to achieve that we created a script that generated objects for each shell, each equipped with settings particular to that layer. Unity boasts an interface to interactively edit values of scrips in the editor, making it possible to update values for both the objects and the shader in real-time.

To allow for more fine-tuning at run-time we decided to generate the transparency of the texture in the shader, as opposed to having to produce the texture ourselves whenever we want to try a different amount of hair or grass, or different sizes of hair or grass. The color of the generated texture however is decided by uploading a texture of choice, allowing for maximum flexibility of each aspect of the generated texture (see figure 3.15). For the generation of the transparency we used Simplex Noise[25] together with hashed numbers (Integer Hash - III)[26] in order to mitigate artifacts otherwise appearing in the generated noise. When generating the transparency we smooth-step between the values. In essence we build upon the random pattern created using the Simplex Noise algorithm, meaning that we use the random pattern of simplex noise for the underlying randomness which is then smoothed. This results in the addition of smaller fuzz of grass between the straws that makes the grass more visually realistic; this creates variation in the straws, as opposed to if the straws were just binary.

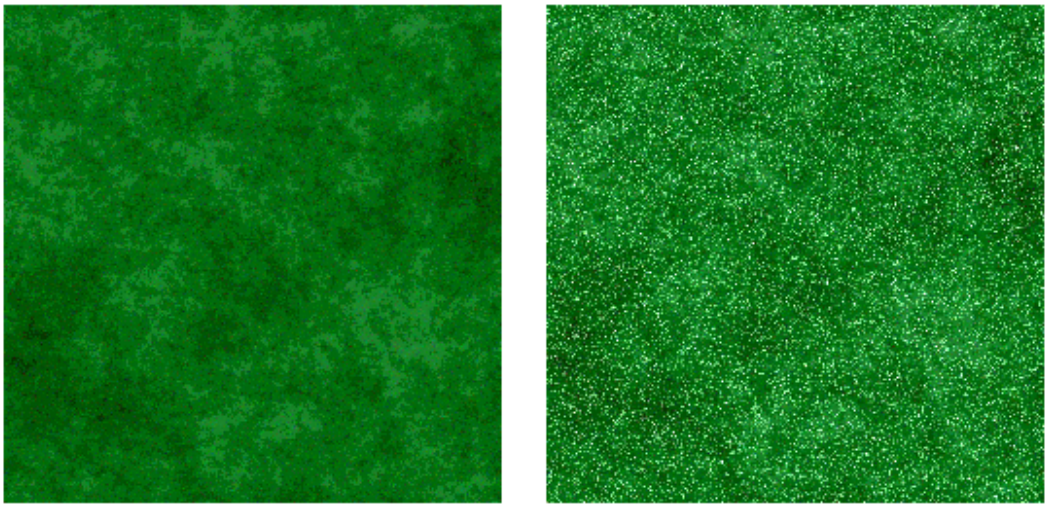

Figure 3.15: The left image shows the uploaded grass texture, and the right image that same texture with an example of the generated transparency applied to it.

At this stage of the implementation, nicely trimmed grass, akin to that of a well tended golf course, is achieved. However, as we are aiming for a natural more forestry look, we had to come up with a way to further develop the method presented by [22], in order to support length variation in the grass. We extended the method by generating of a second layer of noise (with lower frequency than the noise used for transparency), this time again using Simplex Noise. We used this layer of noise to form cavities in the grass, essentially switching color for transparency at certain parts on certain layers, according to the noise. This visually results in grass tufts with varying length (see figure 3.16).

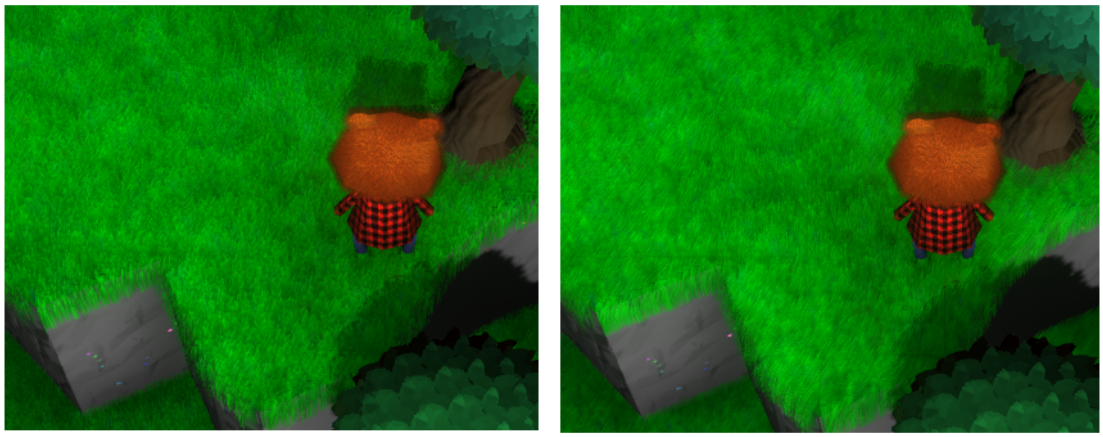

Figure 3.16: Left picture shows grass using noise for both transparency and length variation, whereas the right picture does not apply length variation.

When assigning the final color, the shader checks what layer is being computed. This is important in order to have different settings on different layers. For instance, the inmost shell should always be opaque. In the script generating the shell objects a layer number is included to be used in the shader. In the shader we use a step function to determine which layer we are on, to be able to make the appropriate computations.

Having knowledge of what layer is being calculated is also important for internal shadowing between the grass straws. Since the grass straws are not geometrical objects we had to solve that in a way that make it seem like the grass straws are actually casting shadows among themselves. As is suggested in [27] you could achieve real shadows by rendering an offset shadow version of each layer, in addition to the normal layers. As we want to keep unnecessary computations to a minimum, we chose another method. The way in which we solve the problem is by exponentially shadowing the layers the closer to the ground they are. This creates an effect in which the grass/fur is brighter and less shadowed the more visible it is, and consequently darker and more cast in shadow the more the "surface" is in between other grass straws or hair strands.

3.3.2 Grass and Fur

The parameters on the grass and fur have different values, that each solve different problems. As for the appearance of the individual straws we have values controlling the size of the individual straws and the density of straws. The latter is needed because when the size of the straws changes, so does the number of straws. This is because we have defined it as a multiplier, i.e. the frequency in the noise function. The frequency is much higher for the fur (since hair strands are generally much thinner than grass straws). The density, or what could be viewed as the coverage, of grass is less than that of the fur; as is also natural when comparing the characteristics of grass to hair.

The texture plays an important part in the final look of the grass and fur. The geometric properties are very important, and are what gives the grass its' style. However the color variation, contributed by the texture, gives the grass more perceived detail. As you can see the in figure 3.17, the bottom image, which utilizes an applied texture, possesses much more color variation than the top image, which does not.[1] As a result, a more natural and imperfect life-like look is achieved.

[1]The reason why there is still visible color variation (despite not using a texture) is because of the internal shadowing between the shells mentioned in section 3.3.1. Without it you would barely be able to make out individual grass straws in the top image.

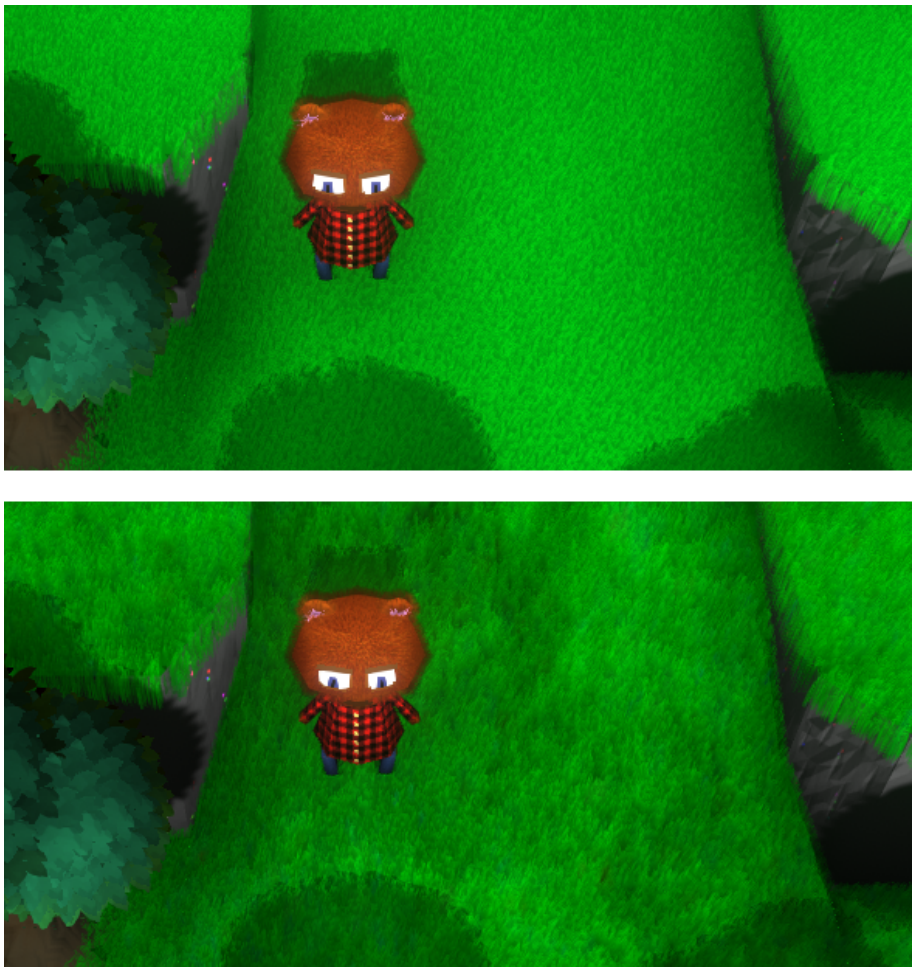

Figure 3.17: Bottom image shows the grass with an applied texture, and the top image without.

3.3.3 Wind

To make the grass, and the world, feel real there needs to be added animation. As a viewer you expect subtle movement in the grass, showing that there is wind that affects the world, and that you are indeed looking at a real 3D simulation of grass. Again, as our grass does not consist of geometrical models, we cannot solve the problem like you conventionally would.

The wind is achieved by offsetting each layer horizontally a little bit, dependent on how the wind is moving and which number of layer it is. The amount which each layer should be offset by depends on how close it is to the ground. If the straw (layer) is close to the ground it will not be bent (offset) by much, meaning the further out the layer is the more it will be offset by. The amount by which they are affected follow the same exponential model as used when calculating the internal shadowing, mentioned in section 3.3.1.[2] The animation of the offset itself is done by using a

[2]The more a layer is offset by, ambient lighting is added to further underline the wind effect, and simulates light hitting the straws.

sine function.

Since the wind on the grass is calculated individually on each tile in the world, having seamless wind across the entire scene becomes a problem. If the tiles' local coordinates are used to calculate the wind, each tiles edge will be a seam. The tiles do not need seams (as it could be viewed as an unfolded bed sheet) so if instead each tile's shader's positional data is offset by the world position of the entire scene, the seams disappear naturally. This solution is not possible for supporting wind on the player characters fur however, as its texture is not seamless, and thus the wind would behave unnaturally.

In addition to just offsetting the layers we also increase ambience on the parts which are offset the most, to give an illusion of light hitting the straws from the side. This gives a stylized appearance, but also incorporates realism in that sense. We do not however add ambience on parts which are in shadow as that would break the illusion.

3.3.4 Masking

Different layers sometimes require different properties (as mentioned, the inmost layer should always be opaque) However there are also the case of layers which call for varying properties throughout that particular layer. One clear-cut example of this is the player character; only the player character's head should be clad in fur, and the characters facial properties such as eyes, should be opaque (see figure 3.18). The way in which we provide this information to the shader is by utilizing the RGB-channels in a mask texture, which we include in addition to the main texture[3] (see figure 3.19). In the mask, using the R-channel means there should be no fur (red), so only an opaque bottom layer. Adding in the G-channel means there should be fur, and an opaque bottom layer (yellow). Lastly, adding in the B-channel means there should be an opaque bottom layer and outer layer, and fur in between (white).

[3]In fact there is the mask texture, main texture (which does not include facial features and is used for the bottom layer and middle layers), and the outer layer texture (which includes facial features).

Figure 3.18: Picture of the player character. We use a mask texture for such things as defining that the clothes and facial features should be opaque, and that the clothing should only be one layer thick.

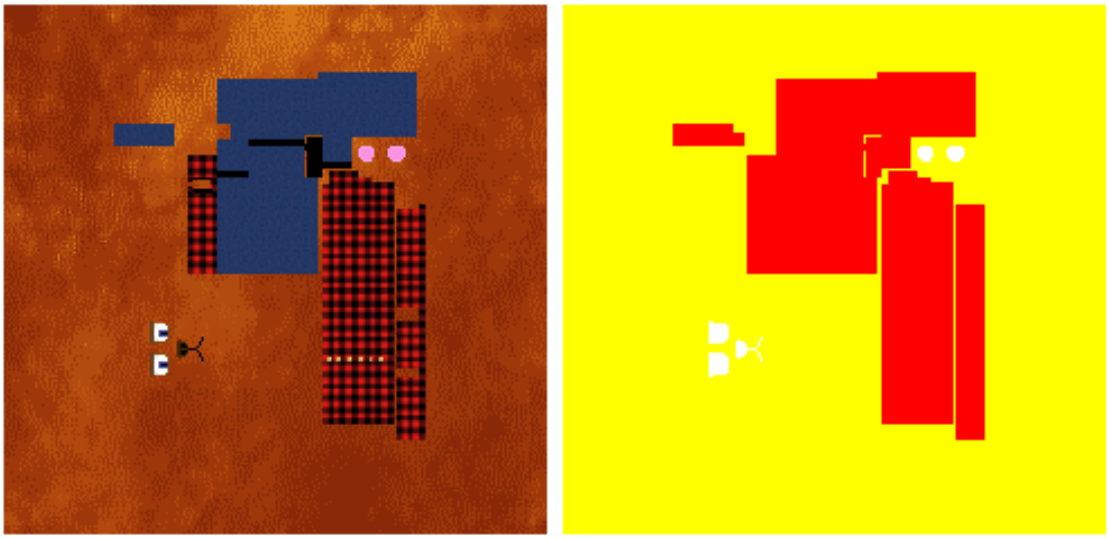

Figure 3.19: Textures of the player character. Main texture to the left, and mask texture to the right.

3.4 Water

Water is an important component in our scene. We decided to create rivers, waterfalls, and an ocean, all of which can be found in every installment in the An-

imal Crossing series. As with the grass we wanted to create something that was lightweight on resources, yet had a realistic, albeit stylized, look.

As the game uses a locked camera view we get some extra freedom in how we can be more efficient in our solution, without it being visible to the player. In many cases when you create water in real-time applications you animate the water with a geometrical model of the waves, or you use normal mapping, or a mixture of it[28]. Both of these methods are more expensive computationally than we wanted, so we explored alternate methods.

For the rivers we use a slowly scrolling ripple displacement texture, and for the waterfalls we use a faster flow displacement texture. This technique is particularly effective for simulating liquids, as water refract and displaces light (due to them acting as shape-shifting lenses). For the rivers and the ocean the displacement simulates small wavy movement in the water, and on the waterfall it simulates a powerful downward flow.

In addition to the displacement, the rivers and the ocean utilizes a solution based on scrolling textures, also used in Nintendo's Super Mario Sunshine[29], and explained in further detail in[10]. This method is employed in Super Mario Sunshine to simulate calm wave movement. We implemented a solution based on this method, but the way we did so differs in the river and ocean shader. Both solutions use two identical diagonally scrolling textures. When these textures scroll over one another they form an animated foggy mess. As you can see in figure 3.20 the resulting texture, from the two layered textures has a gradient from black to white. We only want a small subsection of this resulting texture to be visible on the water, so we have to define rules for this. We therefore set a lower and upper limit to which parts of the texture will be visible.

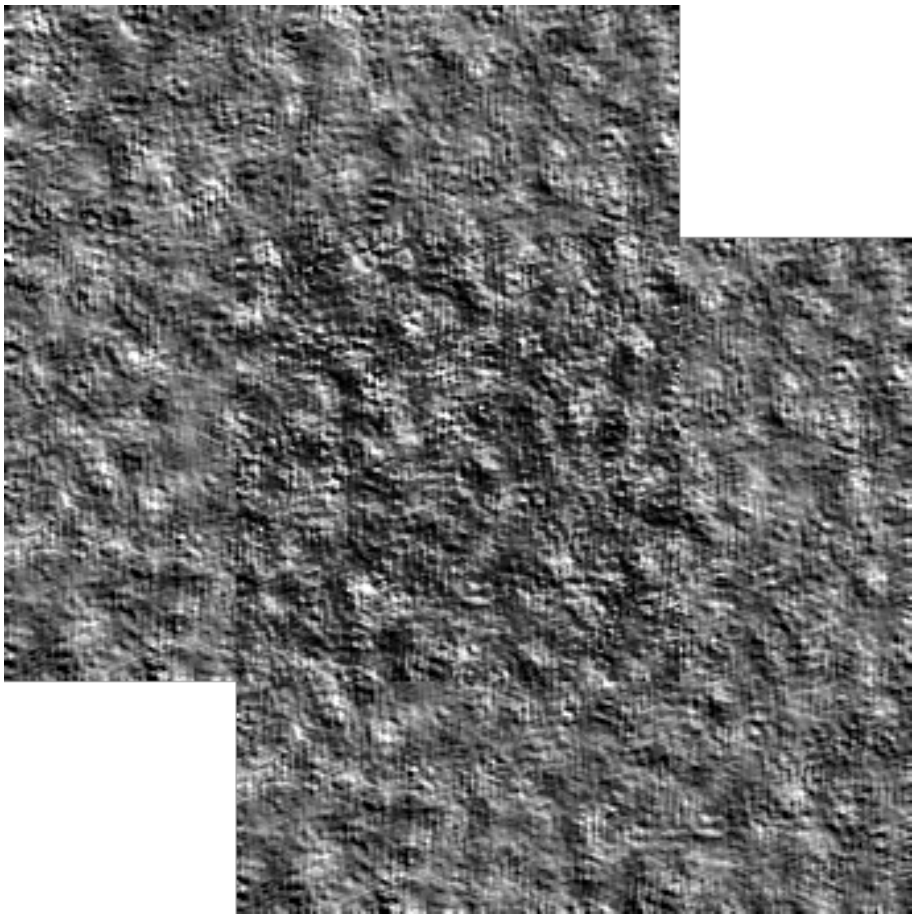

Figure 3.20: The overlap between the two textures is used in the creation of the water.

The values in the resulting texture range from 0 to 1 (white to black). In the river and on the ocean we use a small interval (0.68 to 0.7), since we want thin wave lines. We color this part in a bright color to resemble bouncing light. On the ocean we apply the solution a second time with a larger interval (0.1 to 0.9). We give this part a dark shaded tint and it works to create a 3d effect on the ocean; solving the same problem as the displacement map did on the river, but with different tools.

As far as additional features for the water go there are some notable things to mention. For the river we measure the depth from the camera and write it to a texture in order to create a depth texture. This texture gives us valuable information that we can use in order to add a perceived sense of depth to the water surface. This is not employed for the ocean, as it should look much deeper than the rivers. We also use particle systems for the rivers and waterfalls. On the rivers we have animated sparkles that add some extra style to the water, and on the waterfall we have a mist at the end of the fall to give it a more realistic look.

3.4.1 Water Mesh Construction

The next sub sections will explain the mesh construction of the different water elements.

3.4.1.1 Rivers

When generating the acre meta data as explained in section 3.2.1, some additional meta data is generated that was left out in that section. This data has to do with river generation. All rivers enter from the northern edge of the world and exit through the southern. While doing acre meta data generation, the path of each river is computed as follows.

For each river, the number of which is taken as input to the generator, one of the northern most acres is selected as entry point. Each acre can at most be a single river's starting point. A river walk agent then begins to wander acre by acre, marking each acre as containing a river and its orientation. The river walk agent walks as follows.

The agent can at each step either walk west, south, or east with an equal probability. It can however, only choose one of those directions if their acre has an elevation that is equal to or lower than the current acre. Also, an agent can of course not walk west or east when at the respective outer acres. An agent can walk into another acre containing a river, which will then result in the two or more rivers inside being combined, meaning the agent stops walking as it is done. This walk behaviour will result in acres being marked by rivers that will flow down hill from the northern edge and exit through the southern.

The shape of the rivers has been done trivially, as they are simply straight lines with 90° turns. An acre can be marked with a river in each direction (north, east, south, west), and will consequently contain a straight river from the edge of the acre to the middle for each direction marked. Rivers consist of flat tiles, just as those used for the floor, one for the surface, and one for the river bottom, both of which are offset downwards by a relevant amount. Each river surface tile is also marked with a flow direction, this is so the water can be animated correctly.

3.4.1.2 Waterfalls

The waterfall mesh is simple in theory, but a bit more challenging in practice. When a river passes over a cliff, the idea is to remove all cliff tiles in its path, and replace them with a waterfall mesh. The waterfall mesh is to be connected to the vertices of the two cliff tiles neighbouring the hole created, so that the mesh continues to be seamless and the waterfall adapts to the shape of the cliffs.

Identifying the cliff tiles that the mesh should be connected to, and removing the

ones where the waterfall should be, is trivial with the cliff connection data used previously. The actual mesh placement is a bit trickier. We wanted to be able to model a custom waterfall mesh in an external program, so that it is not just a sharp edged mesh. This meant that vertex manipulation had to be done with care in order to line everything up, since there isn't a one-to-one match between the vertices of the cliff and waterfall models.

In order to achieve a smooth transition between the river and water fall, we constructed the water fall mesh as having the ending part of the top river and starting part of the bottom river. This means the waterfall consists of a flat top part (the end of the top river), a curved fall part (the waterfall), and a flat bottom part (the end of the bottom river). With all these three parts apart of the same mesh, we can blend between visual effects inside a single shader. In order to facilitate this, we utilized custom texture coordinates (coordinates to know where to sample in a texture) as a means of identifying where one effect should begin and another end.

By creating a shader with both the river and waterfall visual effect, we can blend between them as seen in figure 3.21. Finally, we also added particle emitters to the bottom of each waterfall in order to simulate splashing and to hide seams.

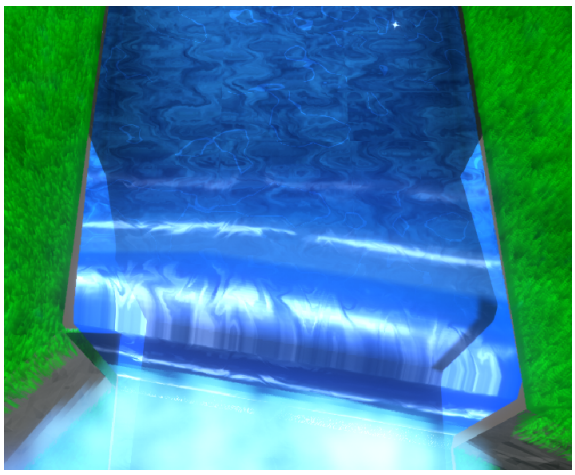

Figure 3.21: A waterfall connected to a river. The transition between the two effects is blended, making it hard to notice where the river ends and the waterfall begins.

3.4.1.3 Ocean & Beach

The ocean and beach mesh construction must be handled together, as it is the combination of the two that creates the wavy beaches that we desire. Both the beach and ocean use the same underlying grid mesh, a grid mesh the size of an acre. The grid mesh is spawned two times for each relevant outer acre, one for the ocean and one for the beach. They are also spawned outside of the world boundaries surrounding the acres just mentioned, this is so that the camera does not see the nothingness outside of the boundaries when in Animal Crossing view mode.

The ocean grid mesh is always completely flat. It animates by moving the entire mesh up and down. The beach mesh however does not animate, it stays static, but is also not completely flat. It uses heightmaps, see section 2.3, for its formation.

The beach heightmap is constructed by the generator in two passes. The heightmap's resolution corresponds to one value per vertex. Firstly, the height of a vertex is determined by its distance to the edge of the world, remember that the beach stretches an entire acre outside the world edges. The farther away, the greater the drop-off. This results in a convex beach surface going lower and lower out to sea, and there is a smooth curvature transition in the corners as well. The second pass, adds an additional downwards offset to vertex heights based on two added sine curves of different frequencies. The sine curves use, for each point, whatever axis that runs along the beach as its input, therefore producing curves in the correct direction. The corners, where the beach transitions from going downwards to sideways, uses the same axis as the western and eastern beaches. This works well, and the corner transitions are rounded and smooth. see figure 3.22.

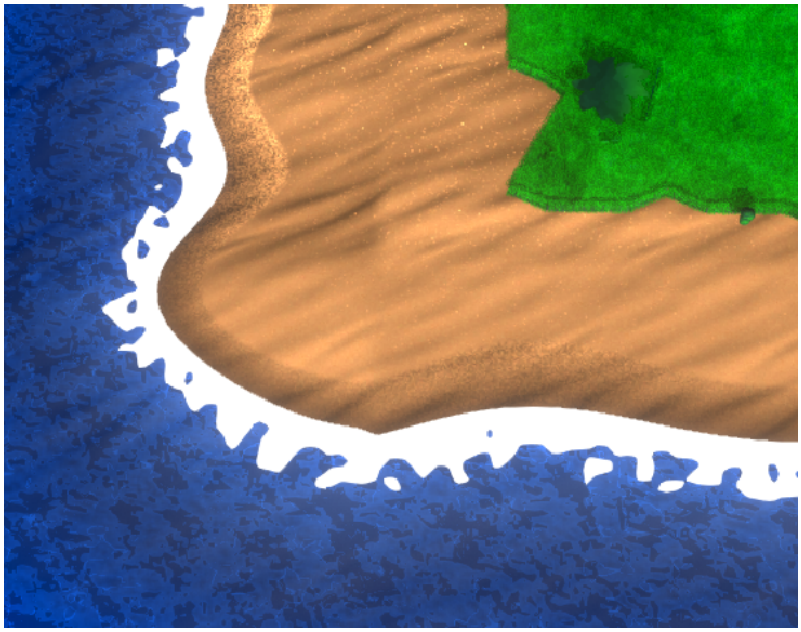

Figure 3.22: A view of the beach, displaying its wavy nature and corner turn.

As can be seen in figure 3.22, the wavy beach effect is a result of the flat ocean mesh clipping the beach mesh. Because the beach mesh differs in elevation based on a sine function in each position, a curvy coast appears. In the figure you can also see a foam line effortlessly following along the water edge. This effect, and others such as the coloring and transparency, utilize the fact that the heightmap can be used as a measure of distance to the water edge. We know the water level and we know the terrain level, therefore we can set some limit in difference between the two where an edge should be painted.

Something that can not be seen in a still image, is the fact that the ocean animates. Waves are simulated

3.5 Decorations

We define trees, rocks, etc. as decorations. The generation of these decorations is highly dependent on the Poisson disk sampling technique presented in section 2.5. Tree and rock meshes are placed in the world using Poisson disk sampling as placement points, where a point is either a tree or a rock dependent on a probability that the generator takes as input. Points which are invalid placement points are discarded before actual mesh creation. Invalid points are those that spawn on top of rivers, waterfalls, slopes, the beach, or too close to cliffs. In order to check whether a point is to close to an invalid tile, a radius is defined for each object. In the case of the tree, two radii are defined one for the canopy and one for the trunk.

Another aspect of the generator that utilizes the Poisson disk sampling algorithm is dirt patches. They are rendered as part of the grass rendering, utilizing the masking technique discussed. However, their placement is dependent on Poisson disk sampling, and equivalent removing of invalid placement points.

In order for the grass shader to know when to apply a mask texture or not, the center points of dirt patches is sent to the grass shader as per vertex data. In order to minimize computation time, not all vertices receive all dirt patches, as this would require unnecessary evaluation of the closest dirt patch. Instead, since dirt patches are static in their location, the closest dirt patches to each vertex are calculated by the generator and stored respectively. Because dirt patches are allowed to overlap in our program, to emulate those found in Animal Crossing, the two closest dirt patches evaluated by each vertex. More dirt patch evaluations can be added as necessary if more overlap is needed.

Figure 3.23: A top down view showing the Poisson disk sampling distribution in effect.

3.6 Dressing

Finally, some cliffs were added surrounding the western, northern, and eastern edges of the world in order to encapsulate the ”play-area” of the world. This mesh is quite straight forward in its construction. The only adaption it makes is its elevation based on cliff positions, size depending on map size, and waterfall placement dependent on rivers. The mesh construction is simply placing flat cliffs by iteration depending on size, and adding waterfalls where needed. The end parts next to the beach are hard coded to turn outwards. The surrounding cliffs can be seen in figure 3.24.

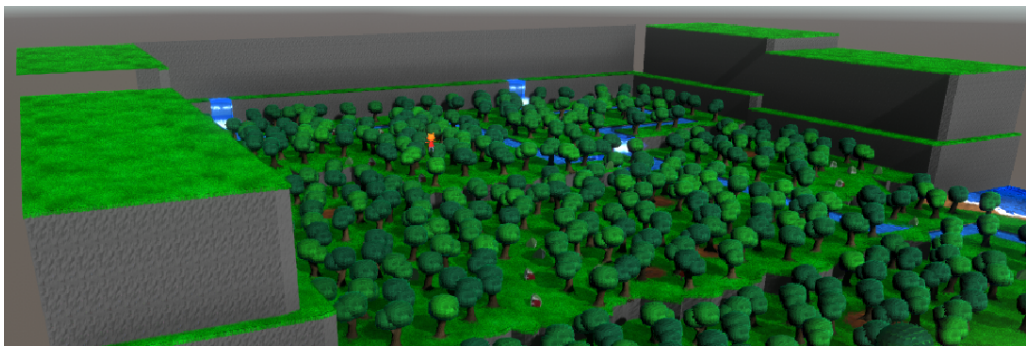

Figure 3.24: A view of the surrounding cliffs.

4

Results

Here we will present the results of our work. First we show a comparison of what features we were able to implement into our generator that correspond to features in the Animal Crossing games. A table is provided, see table 4.1, that gives an overview of the different features, together with images and descriptions of how the implemented features differ from those in the referenced games. Next we show the performance of the generator for different settings. The most crucial stat being load times, however runtime performance is also shown in order to determine the generated content's ability to run in real-time. Finally, we showcase the results of our generator with a compilation of images taken from the produced results together with descriptions of standout features.

4.1 Features

Table 4.1 shows a shorthand overview of the features implemented and missing from those in the Animal Crossing games. A more detailed comparison of each feature is also given.

Feature	Animal Crossing Games	Thesis Project
Surrounding Cliffs	Yes	Yes
Single Elevation Cliffs	Yes	Yes
Beach Cliff	Yes	Yes
Cliff Integrated Slopes	Yes	Yes
Ocean & Wavy Beaches	Yes	Yes
West & East Beaches	Yes	Yes
Rivers	Yes	Yes
Waterfalls	Yes	Yes
Staircase Waterfalls	Yes	Yes
Dirt Patches	Yes	Yes
Decorations (Trees, rocks, etc.)	Yes	Yes
Lakes	Yes	No
Bridges	Yes	No
Multiple Elevation Waterfalls	No	Yes
Multiple Elevation Cliffs	No	Yes

Table 4.1: List of features included and not included in generation.

The next sub sections, highlight each specific feature implemented more intimately, showing comparisons to its equivalent implementation in the games. It is hard however to grasp the entire impact of a feature in a single cropped image. Later in section 4.2, we showcase examples of different generated worlds in a more complete view. Those figures are a good complement to the following subsections.

4.1.1 Surrounding Cliffs

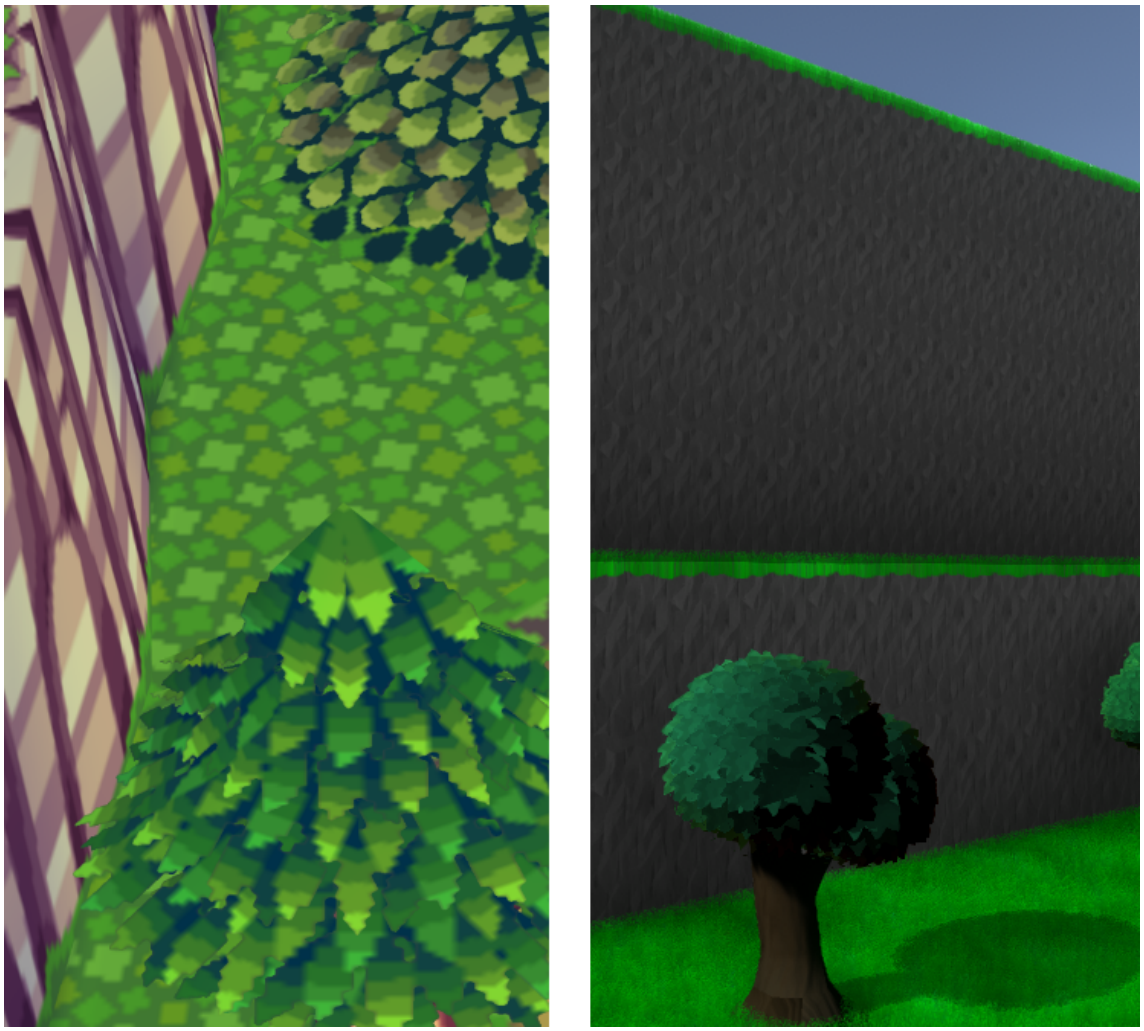

Figure 4.1: The surrounding cliffs encapsulating the world area. Animal Crossing: Population Growing (left) and the thesis project (right).

All Animal Crossing game worlds are bounded in some way. For our generator we mostly took inspiration from the first game, adding straight cliffs at the edges of the map. The Animal Crossing cliffs have a slight form to them, while ours remain completely straight.

The surrounding cliffs ensure that the camera from the players perspective can not see out side the world into nothingness. In our case, we chose to add quite high cliffs, with certain elevation changes in order to make the free fly camera view more appealing. This ”extra” bit of cliff is not needed for the normal Animal Crossing view camera.

4.1.2 Cliffs

Figure 4.2: Cliffs from the first Animal Crossing game (left), latest (top right), and ours (bottom right).

As mentioned in the introduction, and as can be seen in figure 4.2, the cliff formation has changed from the first game to the latest. The first game featured a more free form formation, while Animal Crossing: New Horizons has opted for the more homogeneous straight and diagonal pieces.

We aimed to create a method which could generate the more free form shape of the first game. This can partially be seen in the above figure, but more figures showcasing the completed terrain with the cliffs are available in section 4.2.

Another feature to highlight, is that we chose to allow terrain where cliffs can rise multiple elevations at once. This has not been featured in previous Animal Crossing titles. This is partly why we chose a smaller elevation increase per level than the Animal Crossing: Population Growing. More images highlighting the cliffs of the generator can be found in section 4.2.

4.1.3 Beach Cliff

Figure 4.3: Beach cliffs, the transition from grass down to the beach, from Animal Crossing: New Horizons (left), Animal Crossing: Population Growing (top right), Thesis project (bottom right).

Each Animal Crossing game has featured beaches, and thus also a transition from the main terrain down onto the beach. We call this transition a beach cliff. In the Animal Crossing games, a smooth continuous slope is most often present. In our generator, standard cliffs double up as beach cliffs where the beach cuts it off much higher up, creating a small 90° ledge down to the beach. The beach cliff follows the same formation as the cliffs, giving it a more natural formation compared to straight edges.

4.1.4 Cliff Integrated Slopes

Figure 4.4: Slopes from, the thesis project (top left), Animal Crossing: Population Growing (bottom right), and Animal Crossing: New Horizons (right).

Figure 4.4, shows slopes from both Animal Crossing: New Horizons and Animal Crossing: Population growing. It is difficult to showcase in a single image the true difference in the different games approaches. However, in our analysis, we found the first games slopes to feel more naturally integrated into the cliff walls, while Animal Crossing: New Horizons slopes are more ”neat”. The main inspiration for the slopes of our generator was that of the more integrated slopes of Animal Crossing: Population Growing.

Our slopes can vary in width and length, and try to produce an integrated look which depends on the underlying cliff formation. As in the first game, our cliffs are half ”inside” the cliffs and half spill out, creating a smooth slope to transition between elevation levels.

4.1.5 Ocean & Wavy Beaches

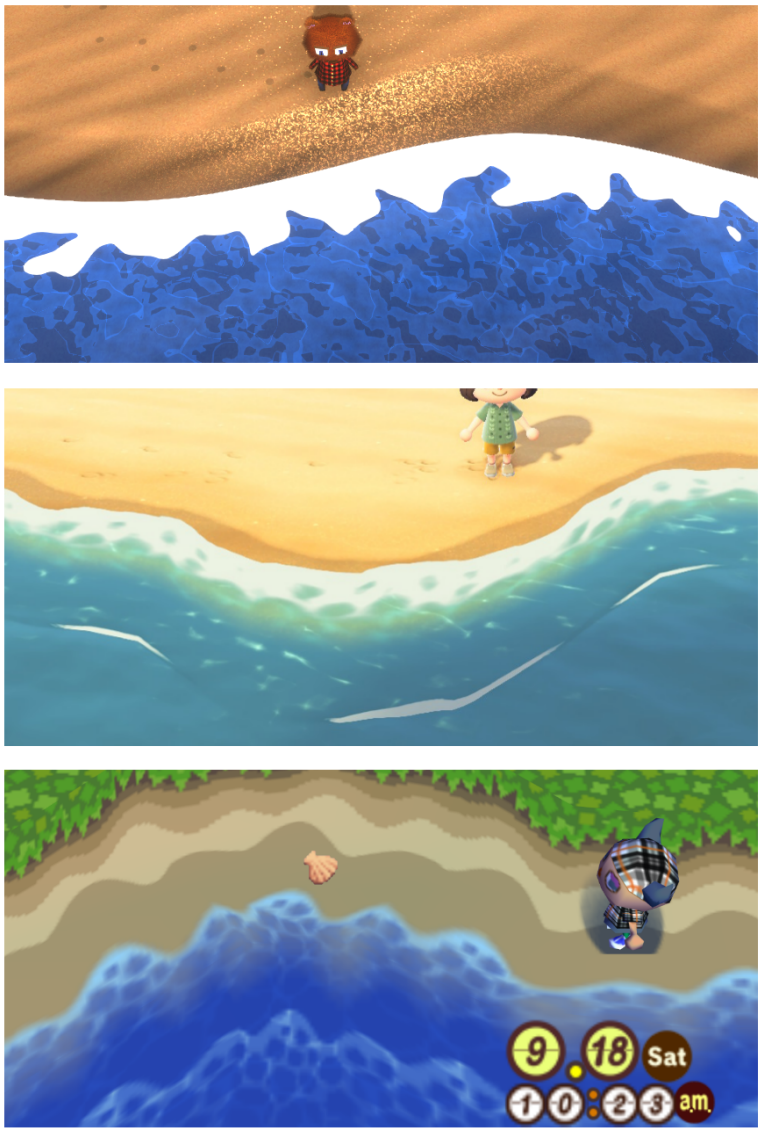

Figure 4.5: Beaches from Thesis project (top), Animal Crossing: New Horizons (middle), Animal Crossing: Population Growing.

Our generator supports the generation of beaches, which have been a staple in the Animal Crossing franchise since the beginning. As can be seen in figure 4.5, the generator produces beaches with similar wavy formations as those seen in the games.

The beaches in our project feature a wave animation, where waves will flow in and out in a wave like manner. Foam will spread out and retract in correlation with the wave animation, and the sand which gets covered by water will darken in tone and increase reflective points to emulate a wet look. The beach will slowly dry until the next wave crashes down.

4.1.6 West & East Beaches

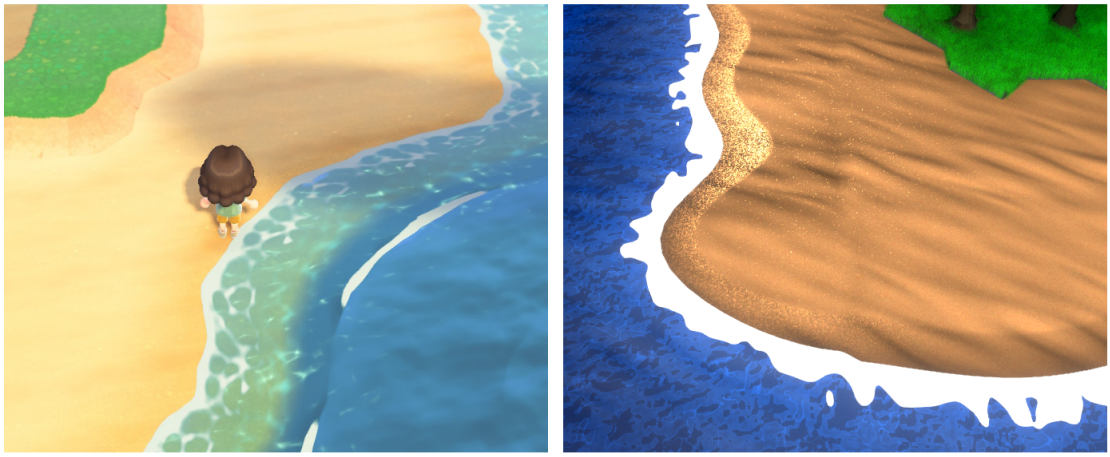

Figure 4.6: A beach transitioning from vertical to horizontal orientation. Thesis project (left), Animal Crossing: New Horizons (right).

Beaches were only present on the south end of the world in the first Animal Crossing game. Later in Animal Crossing: New Leaf, they introduced the possibility for beaches to exist on the western and eastern edges of the map as well. We went with a hybrid solution, where a continuous beach will generate going from the western to the eastern edge, following the southern edge in between. The western and eastern beaches will not generate the entire length of the edges, but rather stop when reaching an elevation change. This differs from the Animal Crossing games, which choose to instead generate the beaches along the entire edges, doing so outside of the normal, elevation changing, terrain area.

As can be seen in figure 4.6, our beaches handle edge changes similarly to those in the Animal Crossing games. A smooth rounded transition.

4.1.7 Rivers

Figure 4.7: Rivers from Animal Crossing: Population Growing (top left), Animal Crossing: New Horizons (top right), and the thesis project (bottom).

Rivers have, like many other features, existed in all titles of the series. The first game had more stream like rivers, which were a bit smaller and with a clear meandering flow. Animal Crossing: New Horizons, on the other hand, has larger rivers but with a less natural look due to the standardized tiles. See figure 4.7.

The rivers in our thesis project, are not as developed in terms of shape. They include a water effect with an animation clearly showing the flow direction. As can be seen in figure 4.7, the river always stays in the middle of acres, with 90° turns.

4.1.8 Waterfalls

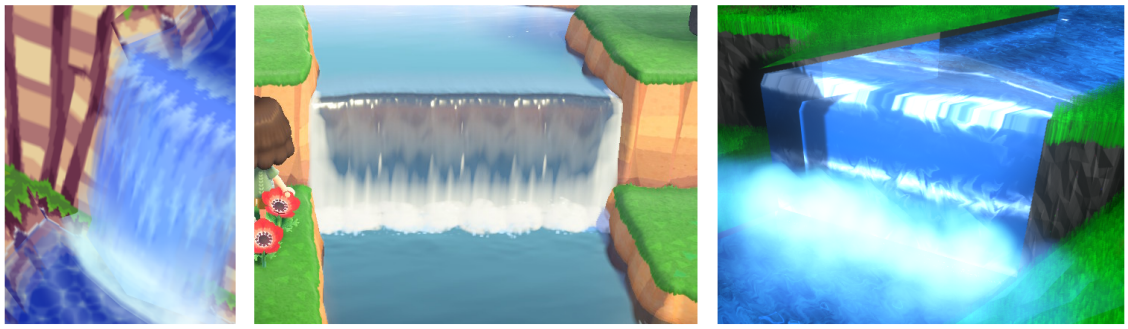

Figure 4.8: Single elevation waterfalls: Animal Crossing: Population Growing (left), Animal Crossing: New Horizons (middle), Thesis project (right).

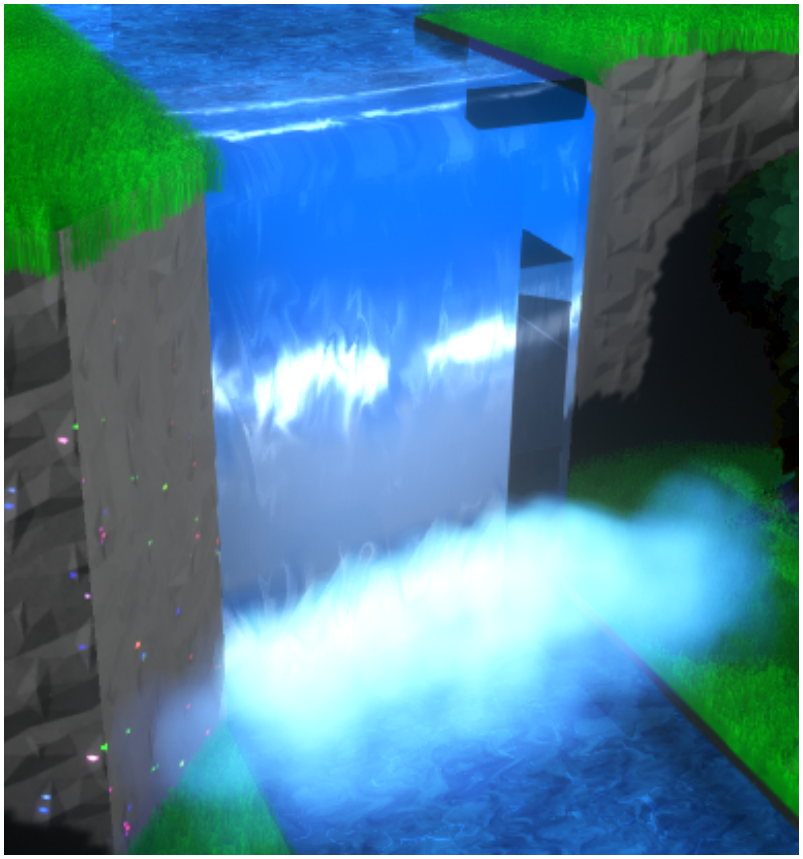

Figure 4.9: Multiple elevation waterfall

Waterfalls is another staple of the Animal Crossing series, present in almost every game. In figure 4.8, you can see a comparison of single elevation water falls from the first and last Animal Crossing titles, as well as from our project.

A feature not presented in any Animal Crossing games, but that is present in ours, is multiple elevation waterfalls. Animal Crossing games have never allowed complete

vertical stacking of more than a single elevation, thus only including waterfalls of a single elevation. However, because we chose to allow multiple elevation cliffs, we also include multiple elevation waterfalls as can be seen in figure 4.9.

4.1.9 Staircase Waterfalls

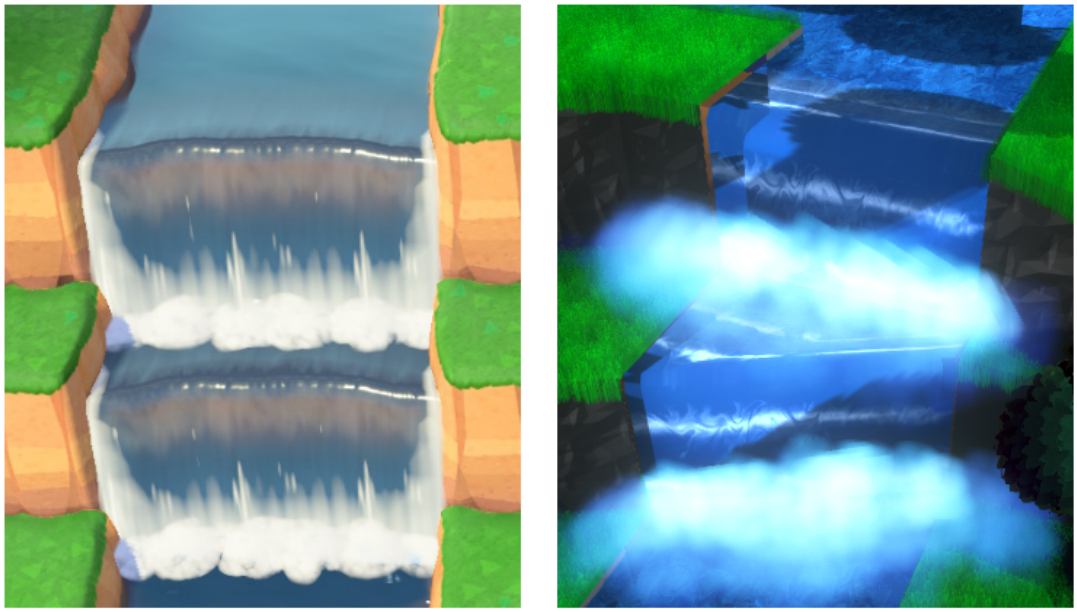

Figure 4.10: Staircase waterfall: Animal Crossing: New Horizons (left), Thesis project (right).

Animal Crossing: New Horizons introduced the ability for the player to modify the terrain to their liking. With that power, players could now do things such as putting two elevation changes one after the other, and add a water fall. Thus creating, what we call, staircase waterfalls.

Our generator also supports the possibility of generating terrain where many elevation changes can happen over a short distance, thus creating staircase waterfalls. See figure 4.10.

4.1.10 Dirt Patches

Figure 4.11: Dirt patch from Animal Crossing: Population growing (left) and the thesis project (right).

In order to break up the repetitiveness of the big areas of grass featured in all the games, dirt patches have been used. The first game, Animal Crossing: Population Growing, had very distinct circular shapes to their dirt patches which we emulated in our generator.

The dirt patches of our generator spawn evenly throughout the world, trying to avoid bad placement, such as on a cliff edge or inside a river. We also allow dirt patch to overlap create a string of circles that may overlap. This is also featured in the first Animal Crossing game.

4.1.11 Decorations

Figure 4.12: Tree and rock decorations from Animal Crossing: Population Growing (left), Animal Crossing: New Horizons (middle), and the thesis project (right).

The Animal Crossing franchise has featured many terrain ”decorations” throughout its time. Most consistently it has always included rocks and trees spread out in the world. See figure 4.12.

Our generator places trees and rocks throughout the world with an even distribution, and helps sell the terrain as being a forest. We achieved a stylized look that hearkens back to the Animal Crossing style, yet is distinctly its own, and that is placed throughout the world in a similar fashion to the games.

4.2 Performance

The performance goals and results are viewed in relation to games, as we view it as the most poignant applicable area. Also, the base inspiration for the generator comes from the Animal Crossing game series. With that said, it could also be used within 3D animations, which are rendered offline and have different requirements.

The performance goals set for this project were to a) have the program be able to run in real-time at interactive frames per second (FPS) when the terrain has completed generation, and b) have the load times of the terrain generation be reasonable for the purposes of a game. We define real-time, interactive FPS, to 10 FPS, but ideally we wanted the program to run at game industry standard of 30, or even 60 FPS.

Though we do show runtime performance numbers, their actual values are not that important. They are mostly to show that the methods outlined in this paper are valid for a real-time program, such as a game. The average FPS during runtime can be viewed in table 4.3. However, the more interesting numbers are the load times.

At the start of the project we defined what a reasonable load time for a game would be, at its highest, 30s, for a one time load. Our generator was able to stay under this limit by a substantial margin, indicating that its use for games is a possibility.

Table 4.2 shows the load times for three different map sizes. The first map size is 5 by 6 acres, with 16 tiles per acre, the same size as the first Animal Crossing game. The next map size tests an unorthodox size for an Animal Crossing world, by doubling the x-axis acres to ten. With this, we also double the amount of rivers spawned. Finally, we also do a test of generating map sizes of double the y-axis as well. These maps were generated with all features enabled. This includes: cliffs, rivers, waterfalls, slopes, decorations, and dirt patches.

Reproduction of these results depend on a number of key factors. First of all, the system specification of the computer the tests were run on are detailed below each table. Another vital point is that the application has been developed in the Unity engine and run through the Unity editor. The application is a standard 3D project in Unity (not a Universal Render Pipeline project), version 2020.3.26f1 (long term support), and uses the default settings for Windows 64 compatibility. No additional

plugins or addons have been used with the project. The git repository can be found at: https://github.com/annlova/MasterThesis. In order to get the git project working, Blender (a 3D modeling tool) must be installed on the computer, as the project uses raw blender files for the models.

For the load time tests, we generated worlds of three different sizes, 10 times each. Each world generated uses a new random seed, as the results should reflect the average expected load time independently from which seed you get. In order to perform the runtime tests, a world was first generated, then we measured the average fps over a 2 minute period free-flying around the world in a semi-predetermined fashion. Then the camera view was switched to that of a traditional Animal Crossing style camera and again we measured a 2 minute period while walking around with our character. Each test tries to capture similar features of the world (beach, ocean, waterfalls, cliffs, forest), but due to the procedural nature of the world, no two tests can follow the exact same path (especially for the Animal Crossing view test).

Measured Time (Avg. over 10 runs)	A	B	C
Total Time	959.8ms	1790.3ms	3391.4ms
Cliff Generation	17.3ms	26.1ms	48.8ms
Floor Elevation	7.5ms	16.8ms	21.6ms
Cliff Floor Elevation	6.1ms	10.7ms	32.4ms
River Generation	63.5ms	110.1ms	216.7ms
Slope Generation	1.6ms	1.8ms	1.7ms
Decoration Generation	52.1ms	107.8ms	243.9ms
Waterfall Generation	7.0ms	7.9ms	8.6ms
Closest Dirt Patch Compute	3.3ms	4.5ms	5.8ms
Terrain Mesh Creation	800.9ms	1504.2ms	2811.4ms

Table 4.2: Load times for different map sizes and number of rivers. Tests were performed on a computer with the following specs:

- AMD Ryzen 9 3900X 12-Core Processor @ 3.8GHz
- Nvidia Geforce RTX 3080
- 64 GB DDR4

A: X = 5, Y = 6, R = 3
B: X = 10, Y = 6, R = 6
C: X = 10, Y = 12, R = 6

X is the number of acres on the x axis, Y is the number of acres on the y axis, and R is the number of rivers generated. The measured time's are averages of 10 generations.

Measured Time (Avg. over 10 runs)	A (FPS)	B (FPS)	C (FPS)
Average FPS (Flying)	278	189	164
Highest FPS (Flying)	530	374	195
Lowest FPS (Flying)	160	88	79
Average FPS (AC View)	311	281	185
Highest FPS (AC View)	481	388	238
Lowest FPS (AC View)	283	267	179

Table 4.3: Load times for different map sizes and number of rivers. Tests were performed on a computer with the following specs:

- AMD Ryzen 9 3900X 12-Core Processor @ 3.8GHz
- Nvidia Geforce RTX 3080
- 64 GB DDR4

A: X = 5, Y = 6, R = 3
B: X = 10, Y = 6, R = 6
C: X = 10, Y = 12, R = 6

X is the number of acres on the x axis, Y is the number of acres on the y axis, and R is the number of rivers generated. The measured time's are taken while roaming a map for a couple of minutes. By flying, we mean that the camera was allowed to fly however it wanted both far and close to the terrain. While AC view, is that of a more traditional Animal Crossing camera, where we follow our player bear character around the island from a static distance.

4.3 Showcase

This section showcases different terrains that have been generated and some stand out features within them. Here you can get a better overview of the terrain in its entirety when generated.

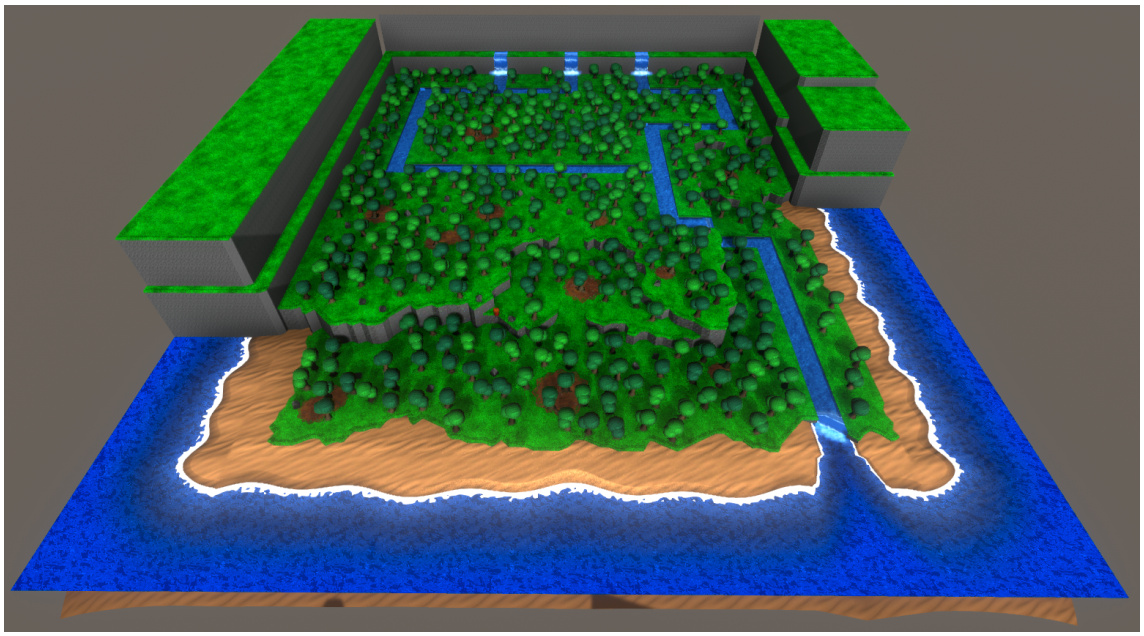

Figure 4.13: Our standard world: 5x6 acres with 3 levels of elevation.

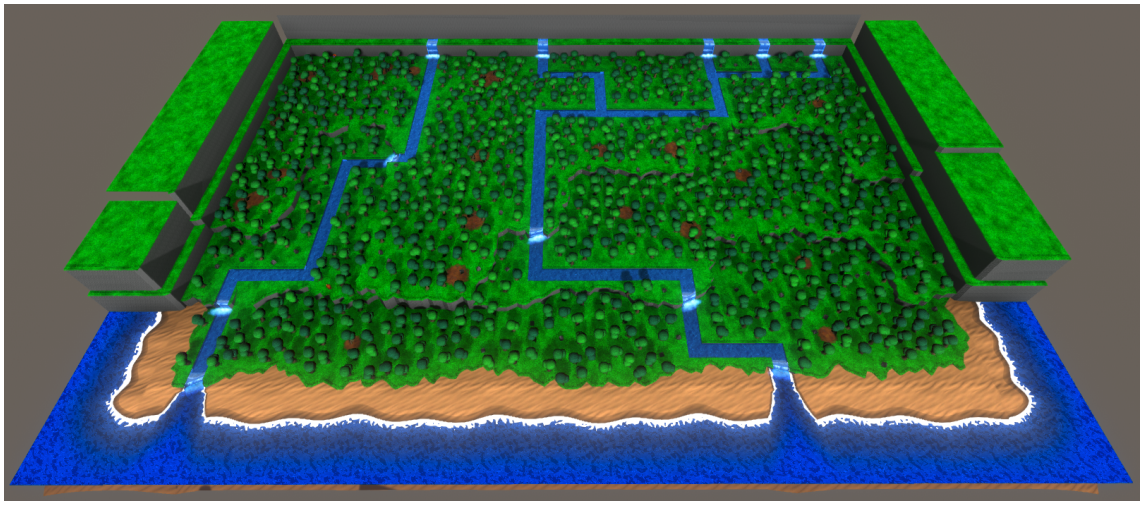

Figure 4.14: 10x6 world

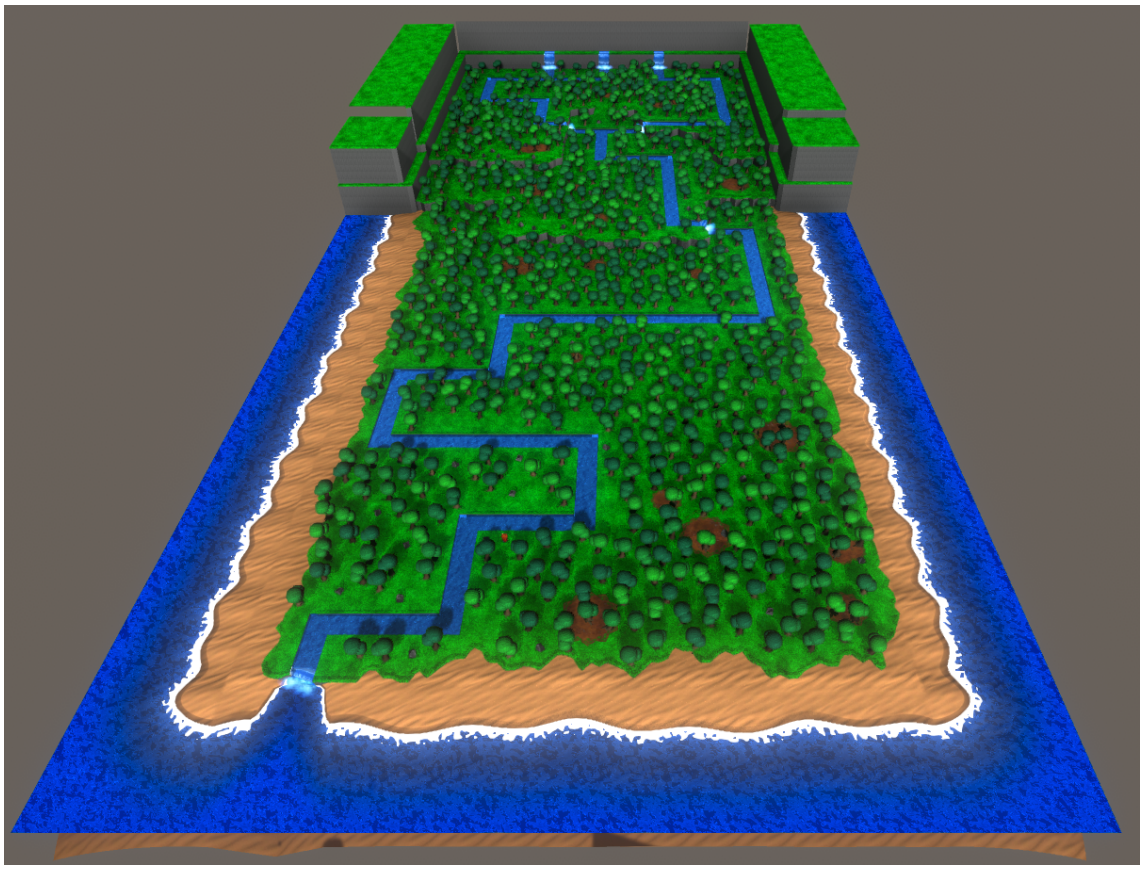

Figure 4.15: 5x12 world

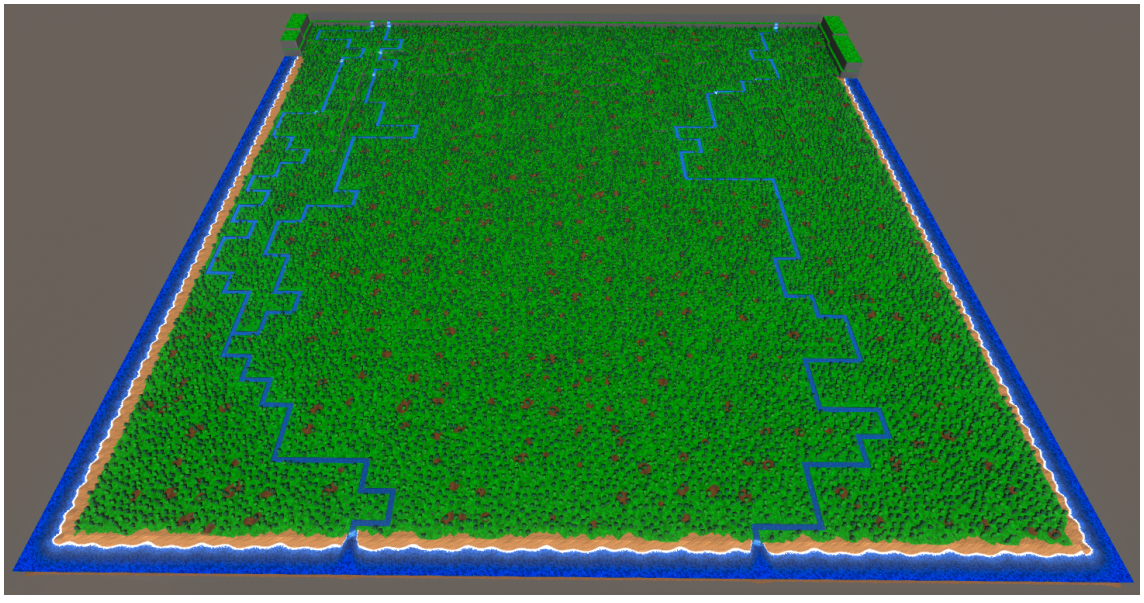

Figure 4.16: 30x30 world

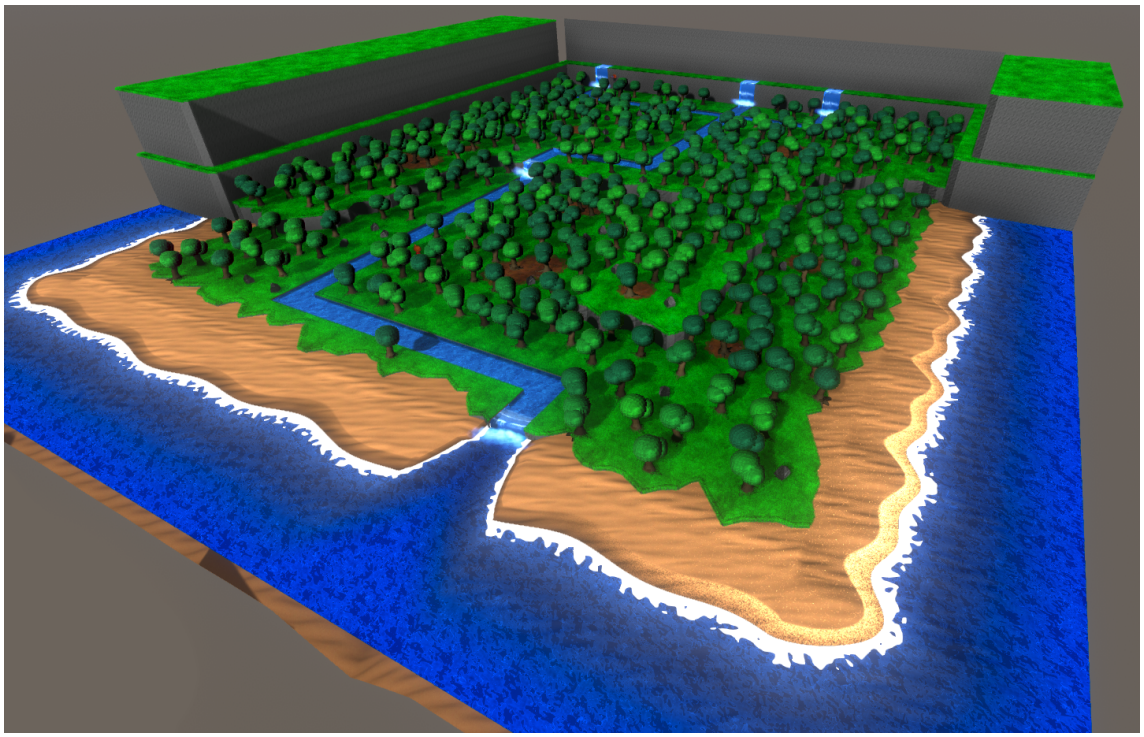

Figure 4.17: 5x6 world with only 2 levels of elevation.

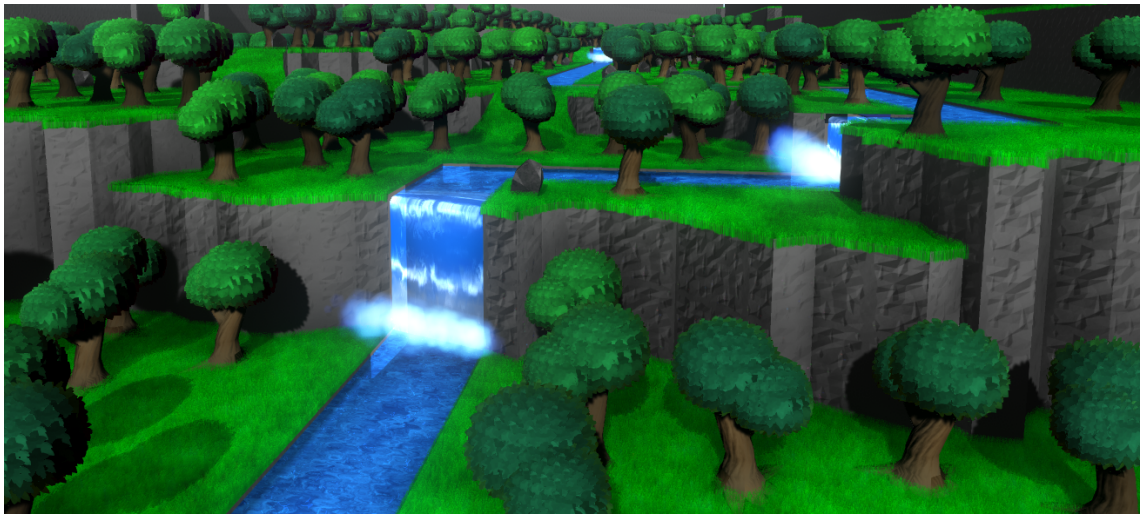

Figure 4.18: We allow for additional levels of elevation.

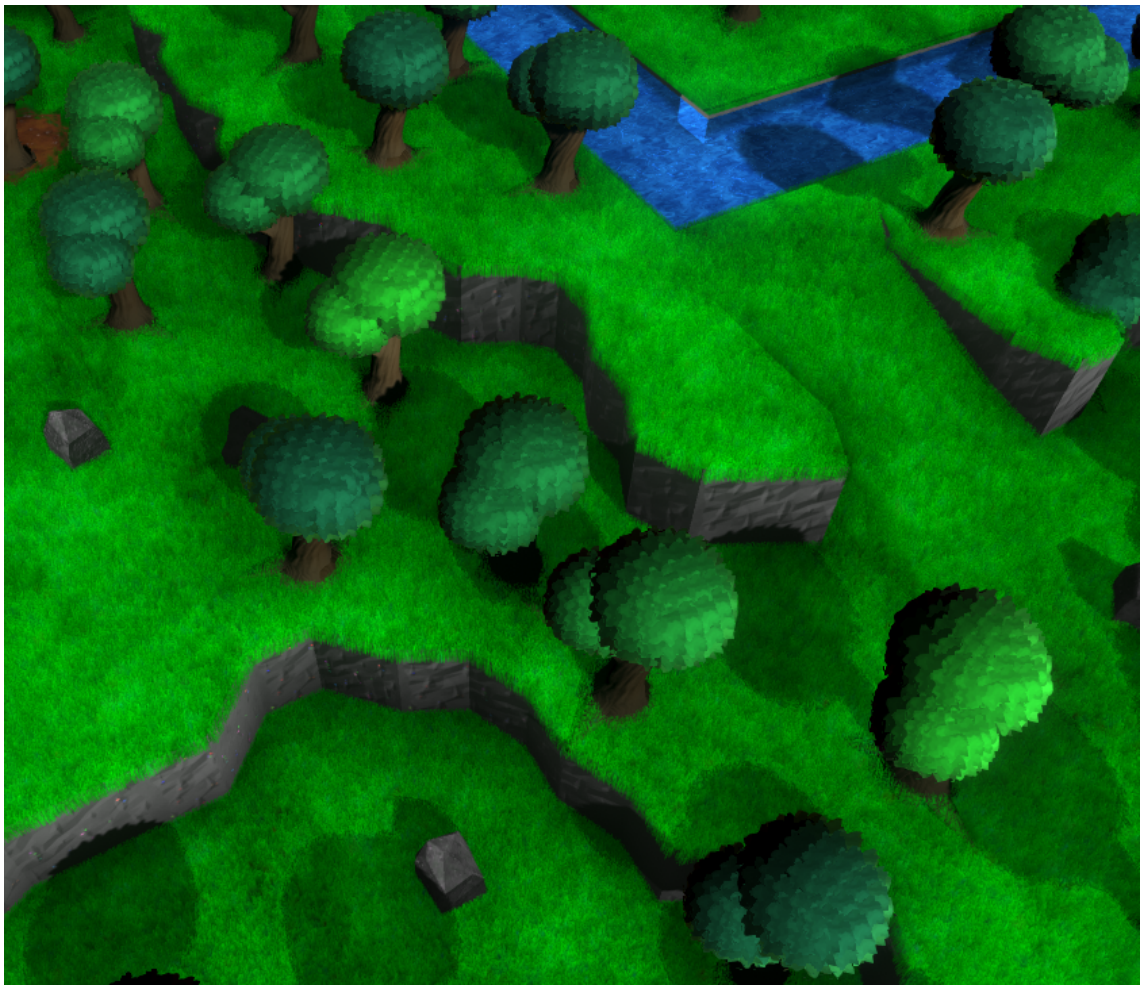

Figure 4.19: Cliff corridor: generated path between two acres on the cliff side.

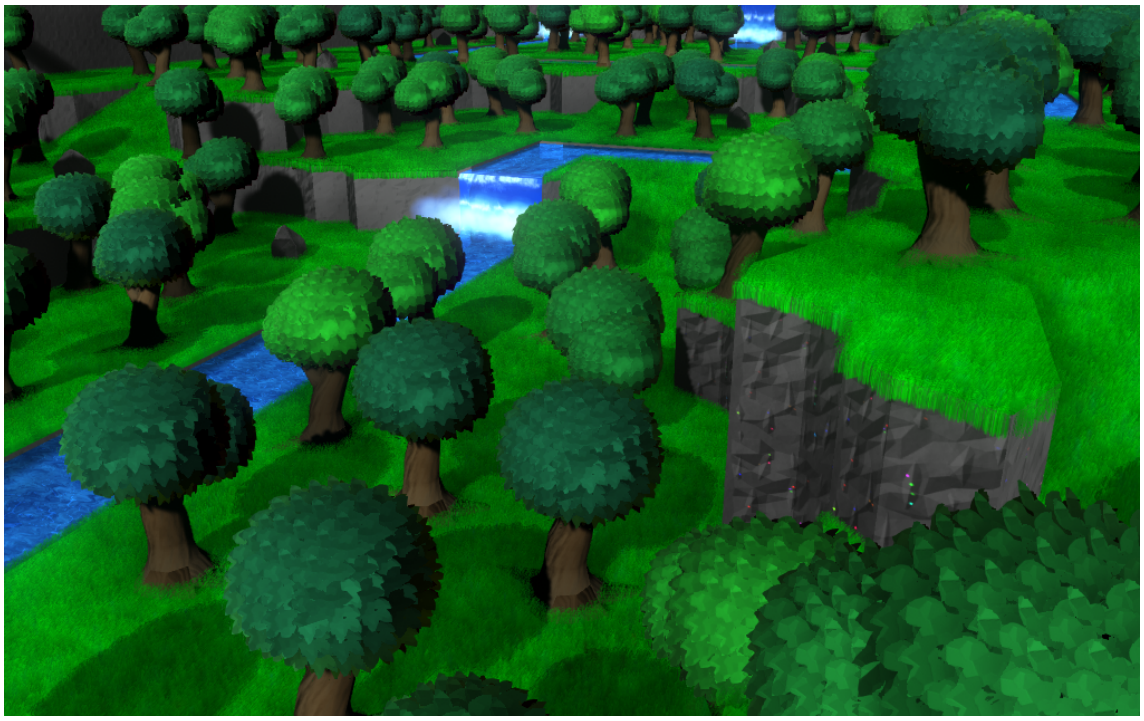

Figure 4.20: From 5x6 world

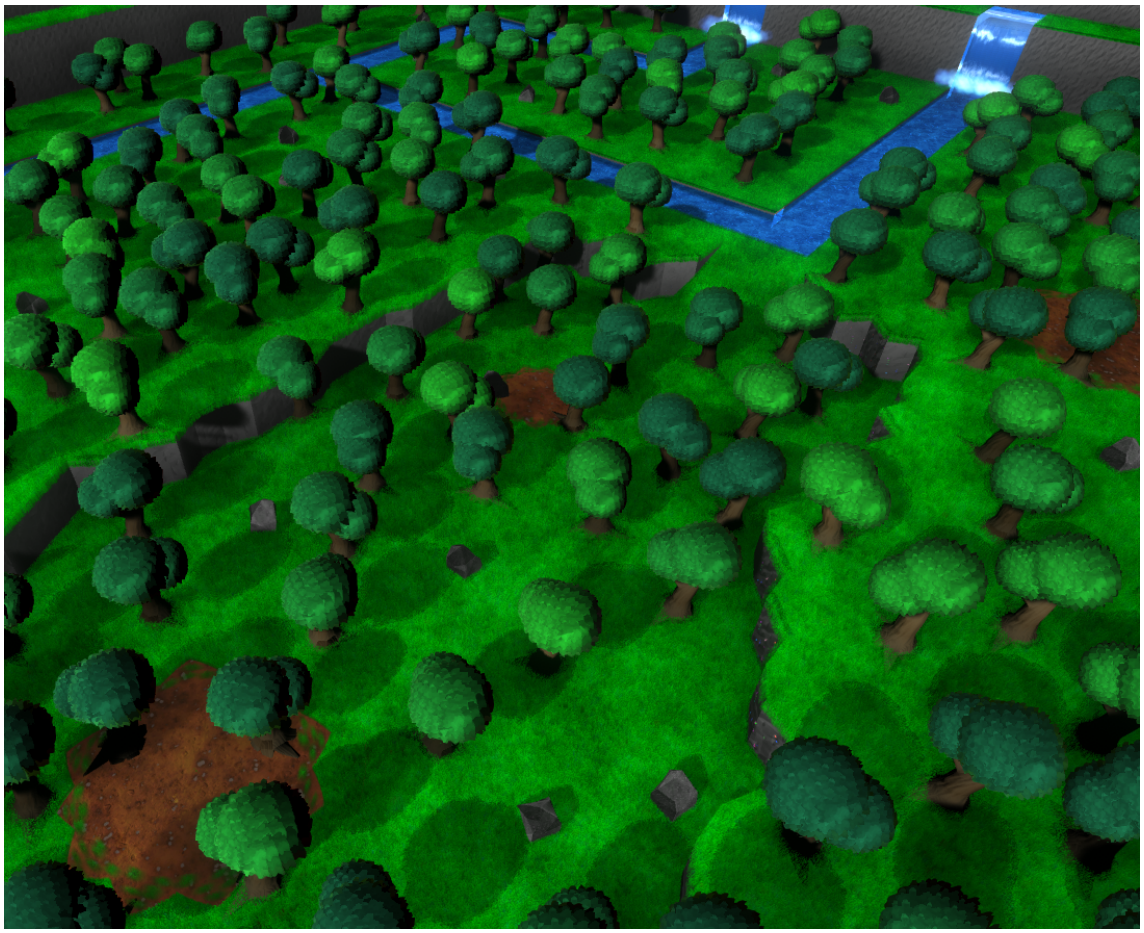

Figure 4.21: From 5x6 world

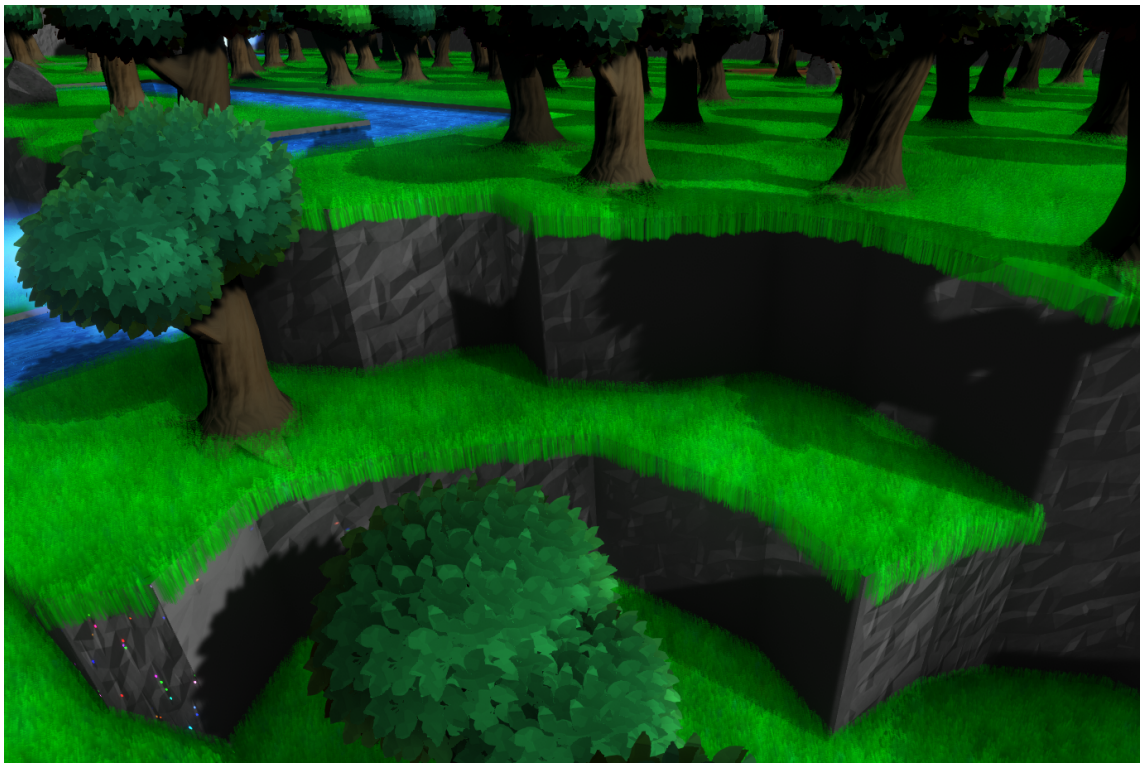

Figure 4.22: From 5x6 world

Figure 4.23: Video showcasing results

5
Conclusion

5.1 Future Work

We have left many features outside the scope of this project that would contribute towards making the game world truly adhere to Animal Crossing's style. Needless to say, gameplay features that belong to the core of Animal Crossing's insignia are missing, since the gameplay loop is not part of our investigation. Additionally, many visual elements are also unimplemented as we have focused on recreating the most integral visuals components within our time constraint.

5.1.1 World features

With the exception of bridges, our demo does not testify of any habitation, such as homes or boutiques. Many things can be added here, but what ultimately depends on how the gameplay is designed. If for example we would like to create a more expansive metropolis area (as seen in Animal Crossing: City Folk and Animal Crossing: New Leaf) we could employ procedural generation to how these areas are generated, to add more uniqueness to each town.

In terms of natural entities such as greenery and rocks, we have only scratched the surface, and much more can be included. Visually Animal Crossing has always kept such entities rather unvaried. All trees, rocks, fruit, and flowers are for the most part identical. Many would probably argue that it is best kept that way. However we can see that the series has experimented with light visual variety on for example weeds, in Animal Crossing: New Horizons. It would be interesting to see how further random variation would affect how the player experiences the game.

When doing our investigation we certainly had an urge to innovate even though we were assessing previously made work. We believe that if we solely copied previously existing features without adding anything fundamentally new about how that feature was implemented, that would not be as beneficial as thinking of alternate ways to solve the same problems. One completely new aspect that could be added to the world, would be caves inside the terrain. This could possibly provide a another layer of possibilities to how new ways that the player can interact with the world.

5.1.2 Visual features

Many notable features such as weather and day cycle has been left out of the investigation. While these would be considered essential for a final product, we believe that creating a standard world (that is daytime at summertime) suffices for the demo. Examining the world during these differences are certainly related to our research at hand. However, since it is time consuming work to add additional scenarios, we decided to narrow down the investigation.

More notably, several core features could be improved. The rivers is one of the most stand out examples, as they are quite basic in our generator. Given more development we would like to see rivers which meander like those in Animal Crossing: Population Growing, while maintaining the width of the rivers introduced in later titles.

Another feature that could be improved is the beach cliff. It would be nice for the beach cliff to emulate that which exists in the games, where its more of smooth slope rather than a ledge.

There are features that were not implemented at all, which would be good future work to expand on the generator. Bridges are most likely quite trivial, however lakes might be more of a challenge. Nonetheless, lakes are a big part of the Animal Crossing terrain, and would be a good addition.

5.1.3 Performance

We are happy with the performance numbers we were able to gather. They show that the generator, has enough leeway to most likely be of adequate performance on different hardware as well, at least in terms of load times.

There was a clear relation between the size of the world and the load times as well as the runtime performance. Load times are good enough to work with even large maps, however runtime performance takes a notable hit. Because we mostly wanted to show off the terrain generating abilities, we used a single big mesh for each feature so that runtime performance would be high when free flying and view the entire map at once. However, if used in an actual game environment it would be more beneficial to divide up the terrain into smaller sub meshes, possibly one per acre, to allow quick occlusion culling of parts that are not shown on screen. When viewing the world from an Animal Crossing style view, only a small part of the entire terrain is seen at once, meaning this addition would most likely improve performance significantly.

The tests were performed in a manner in which they should be reproducible to the extent of the overarching performance results. Due to the inherent randomness associated with procedural generation, there was no one simple exact comparison to measure between different generations. Instead we feel the tests reflect the in-

formation which it truly interesting and valuable in terms of performance. Does the generation load in a reasonable time? Does the application run in a realtime setting? The numbers provided are also valuable from a comparison perspective if future work attempts a similar project to ours. They provide a overview of: using these techniques, you can achieve this type of visual result, at the performance of this. A different project may come up with another way of generation Animal Crossing style worlds, in which they would then present numbers related to performance. Although, these numbers can never be a one-to-one comparison between the projects, there is value in getting a hum for what can be achieved for what cost.

Overall to the project, performance was important to the extent that it showed the viability of the methods in terms of a real-time application. The priority was always on being able to generate an Animal Crossing styled world visually and suitable for interaction. This means that while techniques were looked at from a real-time performance point of view, the application was never ”over optimized” to get the most Animal Crossing per performance unit. Therefore there are most likely a lot of purely optimization improvements that can be done to our generation technique, especially if looking at it from a game perspective.

5.2 Unity

We decided to use the game engine Unity for development. The motivation behind this was principally to be able to swiftly work with a solid and well-optimized foundation, such that we could spend minimal effort on boiler plate matters. We did not have any particular motivation as to why we went with Unity above other engines (such as Unreal Engine) other than having friends and colleagues that had used it for other projects, and other word of mouth.

There have been times when we have felt a bit restricted by Unity, in terms of wanting to do things in ways that we are already familiar with. Fortunately we have found solutions every time that we have been stuck on such matters, so therefore we have never been restricted by Unity’s pipeline in terms of what we want to make. From our time working with Unity we have gotten the impression that it is a very versatile engine, with a large community backing it up, making it a viable option for various kinds of projects.

5.3 Final Remarks

At the start of this paper we ask the question: "How can you generate an Animal Crossing style terrain?". We answer this question defining what the Animal Crossing style is and by showing how to apply specific techniques in order to produce such a result. By this we answer a number of things: Yes, you can procedurally generate terrain in the style of Animal Crossing; Yes, these worlds can be made suitable for

realtime applications such as a game; Applying specific techniques in a certain way combined with some extra details, you can achieve these types of visual results which we believe follow the principles of the Animal Crossing style. Auxiliary to this we also provide insight into ways in which certain aesthetic features can be achieved, which may fit into a world following the style of Animal Crossing terrain.

In answering this question we believe that we have fulfilled our desire to contribute novel knowledge which can be used for real world applications and future research work.

Bibliography

[1] T. Scheuermann, "Practical real-time hair rendering and shading," *SIGGRAPH*, 2004.

[2] H. M, "Poisson disc sampling." [Online]. Available: https://medium.com/@hemalatha.psna/implementation-of-poisson-disc-sampling-in-javascript-17665e406ce1

[3] S. McCombs, "Intro to procedural textures." [Online]. Available: http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures

[4] Unknown, "Town." [Online]. Available: https://nookipedia.com/wiki/Town#In_City_Folk

[5] K. Perlin, "Improving noise," in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 681–682. [Online]. Available: https://doi.org/10.1145/566570.566636

[6] Jessica, "Procedural generation: Creating 3d worlds with deep learning." [Online]. Available: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html

[7] "No man's sky." [Online]. Available: https://www.nomanssky.com/

[8] C. Hutson, "No man's sky steam reviews plummet." [Online]. Available: https://gamerant.com/no-mans-sky-steam-reviews-overwhelmingly-negative/

[9] T. Akenine-Mller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*, 4th ed. USA: A. K. Peters, Ltd., 2018.

[10] Jasper, "How scrolling textures gave super mario galaxy 2 its charm." [Online]. Available: https://www.youtube.com/watch?v=8rCRsOLiO7k

[11] F. Strugar, "Continuous distance-dependent level of detail for rendering heightmaps (cdlod)," *journal of graphics, gpu and game tools*, 2010, https://github.com/fstrugar/CDLOD/blob/master/cdlod_paper_latest.pdf.

[12] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010, https://ieeexplore.ieee.org/abstract/document/5454273.

[13] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 163–, 08 1987.

[14] T. Ju, F. Losasso, S. Scott, and J. Warren, "Dual contouring of hermite data," *ACM Transactions Graphics*, 2002, https://www.cs.rice.edu/ jwarren/papers/dualcontour.pdf.

[15] P. Prusinkiewicz, “Graphical applications of l-systems,” in *Proceedings of Graphics Interface ’86 / Vision Interface ’86*, 1986, pp. 247–253, http://algorithmicbotany.org/papers/graphical.gi86.pdf.

[16] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[17] A. Bundy and L. Wallen, *Breadth-First Search*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 13–13. [Online]. Available: https://doi.org/10.1007/978-3-642-96868-6_25

[18] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, “An adaptive sampling algorithm for solving markov decision processes,” *Operations Research*, vol. 53, no. 1, pp. 126–139, 2005. [Online]. Available: https://doi.org/10.1287/opre.1040.0145

[19] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[20] C. Nunez, “Grasslands, explained,” March 2019. [Online]. Available: https://www.nationalgeographic.com/environment/article/grasslands

[21] K. Bouatouch, K. Boulanger, and S. Pattanaik, “Rendering Grass in Real Time with Dynamic Light Sources,” INRIA, Research Report RR-5960, 2006. [Online]. Available: https://hal.inria.fr/inria-00087776

[22] J. E. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe, “Real-time fur over arbitrary surfaces,” in *2001 ACM Symposium on Interactive 3D Graphics*, Mar. 2001, pp. 227–232.

[23] R. Bridson, “Fast poisson disk sampling in arbitrary dimensions.” *SIGGRAPH sketches*, vol. 10, no. 1, 2007.

[24] K. Perlin, “An image synthesizer,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 287–296, jul 1985. [Online]. Available: https://doi.org/10.1145/325165.325247

[25] ——, “Noise hardware,” 2001.

[26] I. Quilez, “Integer hash - 3,” September 2017. [Online]. Available: https://www.shadertoy.com/view/4tXyWN

[27] bkenwright@xbdev.net, “Fur effects - teddies, cats, hair ....” [Online]. Available: https://www.xbdev.net/directx3dx/specialX/Fur/index.php

[28] C. Johanson and C. Lejdfors, “Real-time water rendering,” *Lund University*, 2004.

[29] J. S. Pierre, “Deconstructing the water effect in super mario sunshine.” [Online]. Available: https://blog.mecheye.net/2018/03/deconstructing-the-water-effect-in-super-mario-sunshine/