NTNU
Norwegian University of
Science and Technology

The Grass is Always Greener: A Procedurally Generated Game

Author(s)

Michael Bråten
Martin Bjerknes

Bachelor in Game Programming
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

16.05.2018

Supervisor Mariusz Nowostawski

Sammendrag av Bacheloroppgaven

Tittel:	**The Grass is Always Greener: Et Prosedyrisk Generert Spill**
Dato:	16.05.2018
Deltakere:	Michael Bråten Martin Bjerknes
Veiledere:	Mariusz Nowostawski
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Michael Bråten, braaten.michael@outlook.com
Nøkkelord:	Prosedyrisk Generering, Programmering, `C#` , Støy, Spill, Unity Engine, Spillmotor, Voxel, Utforskning
Antall sider:	137
Antall vedlegg:	5
Tilgjengelighet:	Åpen

Sammendrag:	Et utforskningsspill i en prosedyrisk generert verden som blir mer ødelagt jo lengre ut du reiser. Målet for spilleren er å nå verdens ende.

Summary of Graduate Project

Title:	**The Grass is Always Greener: A Procedurally Generated Game**
Date:	16.05.2018
Authors:	Michael Bråten Martin Bjerknes
Supervisor:	Mariusz Nowostawski
Employer:	Norwegian University of Science and Technology
Contact Person:	Michael Bråten, braaten.michael@outlook.com
Keywords:	Procedural Generation, Programming, `C#` , Noise, Game, Unity Engine, Game Engine, Voxel, Exploration
Pages:	137
Attachments:	5
Availability:	Open

Abstract: An exploration game with a procedurally generated world, where the world gets more corrupt the further you move. The goal for the player is to make their way through the corruption and to the end of the world.

Preface

We would like to thank:

- Mariusz Nowostawski for being our supervisor and providing us with valuable feedback for both the project and the thesis.
- Simon McCallum for creating the LaTeX template. This saved us a lot of time that we could focus towards writing the thesis.
- Our testers Magnus W. Enggrav, Henrik Bergheim and Joakim Bjerknes for giving feedback on the game.
- The Unity Team for creating the Unity Engine.
- Keijiro Takahashi for making KinoFog, the open source fog effect we used in our game.

Contents

List of Figures

Listings

1 Introduction

1.1 Project Description

1.1.1 Background

The two of us have worked together on numerous projects previously, which is why we decided to do the bachelors thesis together. Before settling on this project we had been offered to join a bigger group for working on a mobile app. We ended up deciding on making our own project from scratch, this way we would have full ownership and control of the product we were developing. We decided to explore the field of procedural content generation. Some of the interesting aspects of procedural generation that appealed to us was the ability to create vast world at runtime in a 3D environment, and the technical challenges that comes with it.

1.1.2 Motivation

We came up with the project ourselves because we have a personal interest in procedural generation. We chose this subject matter because we believe that we have good skills and ideas for getting some interesting results, and that we thought this might prove an interesting project. The reason for why we thought developing this would be fun was because of the iterative process and the emergent characteristics of procedural generation. Some similar projects/games that inspired us are: Fugl [1] and Superflight [2], which are games about exploring a procedurally generated world.

1.1.3 Project Goal

The goal of the project was to create a game revolving around a procedurally generated voxel world. And once we had created the game we wished to publish/release the game in some capacity depending on how proud we were of the end result. If the game became good with nice polish (stuff like graphics, audio, gameplay) we might release the game as a paid download on steam for the price of a coffee or something. If the end result was not as good we would release it as a free download. We also hoped to learn a lot about procedural content generation, noise algorithms and inverse kinematics algorithms. The motivation for choosing this project is to get experience from bigger projects. The subject matter itself was another motivating factor as it let us create a lot of content with few people.

1.2 Project Organization

1.2.1 Academic Background

Both of us are students doing a Bachelor in Game Programming at NTNU in Gjøvik, so we largely have the same academic background. We have experience with Graphics Programming, AI and Game Programming from courses we have had. We have used Unity Engine for game development in some of our previous projects, which is what we will use here as well. The language of choice when developing in Unity is `C#` , which we

have used for some time now. We also have experience designing games from some of the courses we have had.

1.3 Document Structure

1. **Introduction:** Introduction of the thesis.
2. **Game Design:** Describes the initial game design, and how it has changed for the final version of the game.
3. **Requirements:** Non-Functional requirements.
4. **Technical Design:** High level technical design.
5. **Development Process:** Describes the tools and the workflow used during the development.
6. **Implementation:** Explains how we have implemented the features of the game.
7. **Optimization:** Goes into detail on some optimizations we have done.
8. **Usability Testing and User Feedback:** Describes our usability testing with external playtesters.
9. **Deployment:** Explains how we built and published the game.
10. **Discussion:** We discuss the results of the project, the tools and methods used and future work.
11. **Conclusion:** We reflect on the project.

1.4 Terminology

- NPC: Non-player character
- AI: Artificial intelligence
- Shader: A program running on the GPU.
- Mesh: Defines geometry rendered by a shader.
- FPS: Frames per second (frame rate).
- GPU: Graphics processing unit
- CPU: Central processing unit

2 Game Design

In this chapter we talk about the design of the game, both the initial design and how it has changed throughout the project. Having a game design document early on during development gives us an idea of what we want. Even if we don't follow the game design document to a T, we at least have an idea of where we want to end up. It also lets us consider not yet implemented features from the design when creating systems for use in the game.

2.1 Initial Design

In this section, we will go over the initial design we created for the game. The core of the game is exploration of procedurally generated worlds, so all the features of the game is centered around this in some way. We wanted the player to be able to have unique experiences every time they play by letting them explore new worlds every time they play, as well as giving them multiple ways of exploring the world they are in.

2.1.1 World

The world needed to be large, unique and filled with a variety of different terrain so that the player had an incentive to explore the world. To make every playthrough unique, everything in the world is procedurally generated. To make it so the terrain doesn't seem completely bare, we have procedural foliage in the world to cover it.

To evolve the terrain and give the player a sense of progress, we have world corruption. The corruption is an effect that causes the world to grow more unnatural the further the player progresses, breaking the laws of physics by causing deformed terrain and water floating in the sky.

2.1.2 Animals

To give life to the world, we populate it with animals. These animals are, like the terrain of the world procedurally generated, and no two animals should be generated to look exactly the same. There will be multiple types of animals, each having different ways of moving through the world.

Animal Switching

The player is able to control any of the animals in the world. Each animal type plays differently, so that the player will have to swap between the different types to be able to travel through the world. The player has to eat other animals in the world and turn in to them to progress.

To encourage the player to switch animals, we also needed a way to encourage the use of each animal type. Different weather effects, strong winds and storms are used to slow down or prevent travel by air, and heavy snow is used to slow down the travelling on ground. Slowing or restricting water travel is something we didn't think was necessary, as we felt that traversing through water would likely be the least preferable method of

travel. This could obviously be subject to change if we find that this is not the case.

Movement

As we have multiple types of animals, each type had to have its own way of moving around. There will at least need to be three types of animals, some that travels on ground, some in water and some in the air.

2.1.3 Goal

All games need some sort of a win-condition, so we created a goal for the player to reach for. The goal for the player is to reach the end of the world. As it is an exploration game, how they reach the end is up to them. If they want to collect as many animals as possible, want to see as much of the world as they can, or just want to try get to the end of the world as quickly as possible, they should be able to do it the way they want and at their own pace.

2.1.4 User Interface

The user interface had to be simple and easy-to-use. As we want to keep the player's attention at the game, the UI needed to be non-intrusive so as to not take too much attention from the actual game.

2.1.5 Visual Design

The visual design of a game is very important for forming an identity, and for setting an atmosphere for the player. We decided to go for a fully voxel-based world. Terrain, foliage and animals are all made out of voxels.

2.1.6 Audio Design

Audio is not the main focus of the project, but audio does have a part in setting the atmosphere for a game. The environmental audio should feel real, meaning that the player should be able to tell the difference between an ocean and a small lake, and between a desert and a forest. The music should give a peaceful atmosphere, but with a bit of adventure in it.

2.2 Final Design

In this section we will go over the things that changed between the initial game design and the final design of the game, as well as new additions. If something from the original design is not mentioned in this section, then we haven't made any changes to the design.

2.2.1 World

Initially we generated all the terrain using the same parameters. As we said in Chapter 2.1.1, we want the world to be filled with various types of terrain. We weren't satisfied with the variety given, so to allow for more variety in a world we use biomes [3] that can differ in how they generate the terrain through some parameters. The biomes allow for the player to get different experiences in different places in a world.

2.2.2 Animals

In the original design we had not settled on what type of animals we wanted in the game, except for the requirement for three types that could travel by land, air and water. In the

end we stuck to these three types:

- **Water Animal:** Swims fast, but is very slow to the point that it is near unusable on land.
- **Air Animal:** Can fly fast, but is slow on the ground and in water. The bird is strongly affected by windy weather, which makes it non-preferable in some areas.
- **Land Animal:** The best animal type for traversing the world on the ground. Just like the bird it is fairly slow in water.

These three animal types match the three types we wanted per the initial design.

Animal Switching

For switching animals, the original idea was that the player would eat other animals to take over them. In the final design we ended up with a system where the player "swaps brains" with an NPC animal to take control over it, this way the player also leave their old body behind when taking over another animal's body.

For the incentives for switching animals, we ended up not adding heavy snow for preventing ground travel in the end. We found that the player would almost always prefer travelling in the air to travel by land if possible because the player could just look at the terrain without actually having to navigate their way through it. Because the player preferred air travel, we did keep the idea of using strong winds in some places to discourage flying all the time. For incentivising use of water animals, we made the strong winds always appear when over an ocean so that the player had to use a water animal to traverse oceans efficiently.

Collecting Animals

The ability for the player to collect animals was not something we had planned from the beginning, but was added on later. The idea is that we want the player to be able to look at what animals they have used during a playthrough. The should be able to look at all the animals, and see how many they have collected of each of the different animal types.

3 Requirements

3.1 Usability

Game controls should be intuitive so that the user can play comfortably with little to no explanation of how to control the characters. The user interface has to be easy to navigate and the user should not have any problems finding their way.

3.2 Reliability

The game should be running without crashing for as long as the user want to use it, as long as the users computer meets the minimum system requirements for Unity 2017.2 [4].

3.3 Performance

The two measurements of performance for this project is the frame rate and world generation speed. On modern computer hardware the game should perform at at least 60 fps, and as the user traverses the game world, they should not be seeing chunks being generated. We would not expect this kind of result on older hardware, but the game should still run, although likely at a low fps and with visible chunk generation on any hardware that reaches the system requirements for Unity 2017.2 [4].

4 Technical Design

In this chapter we will be talking about the technical design of our game. There are 3 major components of the game that we cover here, Unity, the `WorldGenManager` and Animals. Unity is the game engine we used for making the game. The `WorldGenManager` is the most central class for world generation. The animals populate the game world as either AI or player controlled characters.

4.1 Unity

In Unity, all objects in a scene are GameObjects [5]. GameObjects are comprised of one or more components, where at a minimum it must contain a `Transform` component. The `Transform` contains the position, rotation and scale of an object in 3D space. Components are scripts attached to the GameObject, which can contain data and behaviour. For example: if we want a GameObject to play audio, we add a `AudioSource` component that can be used to play audio. We can then let the `AudioSource` play the audio just once or in a loop, or we can add a custom script `AudioController` to the same GameObject to control the `AudioSource`. Using the `gameObject.GetComponent<Component>()` function, any MonoBehaviour script can have access to any component from any GameObject in the scene.

For a script to be attachable to a GameObject, it must inherit from the MonoBehaviour class [6]. The MonoBehaviour class contains several functions for managing the life cycle of the script, such as:

- `Awake()`: Runs when a script is instantiated. Used to initialize local data.
- `Start()`: Runs the first time the script is enabled after instantiation. Used to initialize data that relies on other GameObjects or scripts.
- `Update()`: An update function that runs every frame.
- `OnCollisionEnter()`: Runs when the script's GameObject enters a collision with another GameObject.

4.1.1 Unity UI

UI elements in Unity are special, in the way that they don't contain a normal `Transform` component, but instead has a `RectTransform`. The `RectTransform` contains size and position data for the UI element relative to its parent GameObject in 2D space, as well as rotation and scale.

4.2 World Generation

The `WorldGenManager` is the core class in the world generation system. This class is responsible for all handling of chunks and animals. When new chunks come into range or a new animal is to be generated, the `WorldGenManager` places an order for the new chunk or animal for the `ChunkVoxelDataThread` to handle. When the `ChunkVoxelDataThread` returns a result from the order, the `WorldGenManager` finishes the order by generating

meshes and doing any other work that can not be done in the worker threads. Once the orders are completed, the `WorldGenManager` launches the chunks/spawns the animals into the game world. The `WorldGenManager` keeps references to all the chunks and animals and removes them when they get to far away from the player, it will also remove any unhandled orders it has made where the resulting chunk/animal would be placed to far away from the player.

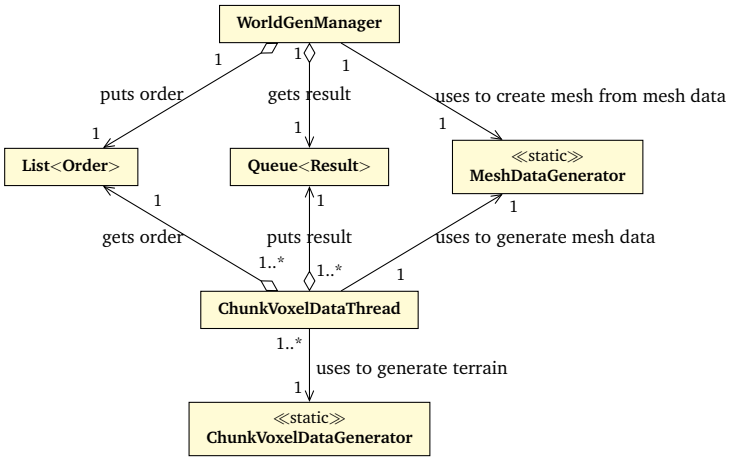


Figure 1: The relationship between the main components in the world generation.

4.2.1 Handling Orders

The `ChunkVoxelDataThread` is responsible for handling the more expensive part of generating chunks and animals. The orders placed by the `WorldGenManager` is handled by multiple instances of the `ChunkVoxelDataThread` class. When `ChunkVoxelDataThread` is to handle an order, it prioritizes which orders to handle based on the player's position and heading. If the ordered item was a chunk, the `ChunkVoxelDataThread` will use the `ChunkVoxelDataGenerator` to generate the actual terrain data. It will also generate any foliage that goes into the chunk. After generating voxel data for the terrain and generating the data for the trees, the `MeshDataGenerator` is used to generate the mesh data for the terrain, water and foliage using the generated voxel data. If the ordered item was an animal, it will generate an `AnimalSkeleton`, and use the `AnimalSkeleton`'s voxel data to generate the mesh data for the animal using the `MeshDataGenerator`.

Once the `ChunkVoxelDataThread` has done the heavy load of the order handling, it sends the result back to the `WorldGenManager`. The `WorldGenManager` will then generate the actual meshes for the chunks/animals using the mesh data that has been generated.

4.3 Animals

The game features 3 different types of animals: `LandAnimal`, `AirAnimal` and `WaterAnimal`, these can be seen in Figure 19. The body of an animal is defined by an `AnimalSkeleton`, which is generated procedurally by the game at runtime (see Chapter 6.2.7). Animals function in their environment, with the ability to run, walk, swim and/or fly depending on the animal type. Animals may also have special actions they can perform, such as jumping or taking off for flight. Animals also have behaviour deciding how the functionality is used. The behaviour of an animal is controlled either by AI or by the player.

The fact that animals can be controlled by either the player or an AI gives us some requirements for our design. We need to separate the functionality of an animal from its behaviour. Functionality is for instance the ability for an animal to walk in a certain direction, whereas the behaviour is the decision to walk in a certain direction.

4.3.1 Initial Animal Design

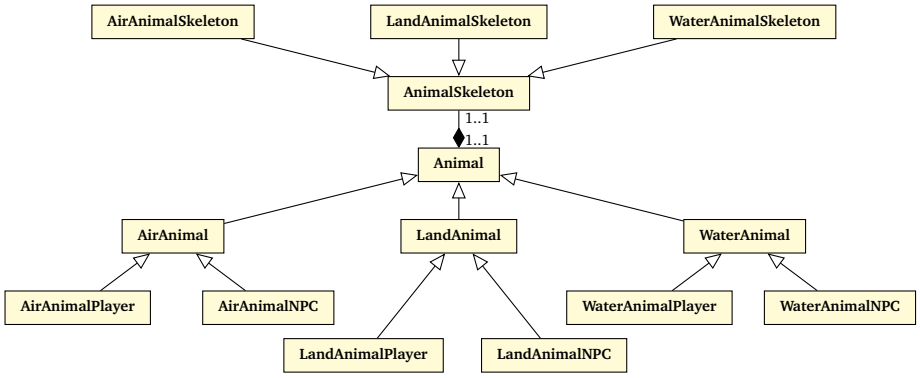


Figure 2: Early animal implementation class diagram

In Figure 2 you can see a class diagram of our first design for animals. Here we implemented the functionality and behaviour of animals through inheritance. The `Animal` class contains base functionality needed for all animals and the `(X)Animal` classes implement animal specific functionality. Behaviour of animals are in this case implemented in the `(X)Animal(NPC/Player)` classes. The body of the animals is implemented through composition. The reason for implementing the `AnimalSkeleton` through composition is to separate the logic for generating an animal from the logic of making the animal functional. The `AnimalSkeleton` class is also quite big in terms of lines of code (as is the `Animal` class) which makes this separation useful for maintainability.

Issues with Initial Design

The biggest issue with the first design is how behaviour is implemented. Behaviour is not sufficiently separated from the functionality of the animal in this case as it is implemented through inheritance. The player can become any animal they find in game if they want to (see Chapter 6.10.1). Becoming another animal requires swapping the behaviour of two animals. With this implementation, swapping behaviour between two animals is not elegant since behaviour and functionality is provided by the same object. This makes it so that we have to destroy and recreate the animals using their previous animal skeletons so that they will look the same afterwards.

Another issue is the lack of inheritance for the behaviour. The inheritance structure in Figure 2 is mostly supportive of the functionality of animals. The animal behaviour also needs an inheritance hierarchy. There is common functionality between the various NPCs and player animal behaviours. With this design we either have to duplicate the common functionality in the various derived classes, or we have to put it in the `Animal` class where it does not logically belong. We could have solved the inheritance issue using multiple inheritance, but `C#` does not support multiple inheritance [7]. Multiple inheritance would also not fix the animal behaviour swapping issue.

4.3.2 Final Animal Design

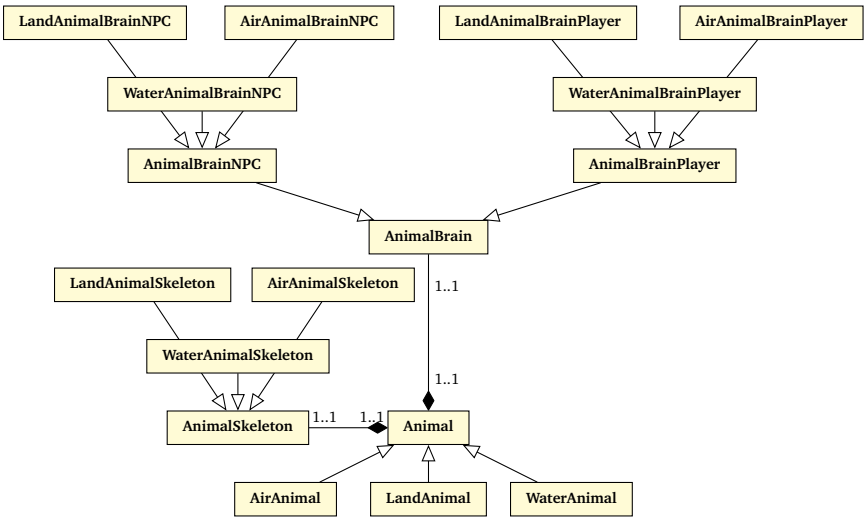


Figure 3: Current animal implementation class diagram

The final design addresses the issues raised for the first design. Animal behaviour gets properly separated from the functionality of the animal. This is achieved by implementing the behaviour through composition instead of inheritance. The behaviour is now implemented in the `AnimalBrain` class. Using composition simplifies the process of swapping the behaviour of two animals, which can be done by swapping the `AnimalBrain` of the two animals.

Since behaviour is implemented in a separate object in this case, we can give that object its own inheritance tree as seen in Figure 3. `AnimalBrainNPC` contains functionality common for all NPCs and `AnimalBrainPlayer` contains functionality common for all player controlled animals. This solves the second issue raised for the first design.

5 Development Process

5.1 Working Hours

For the project we decided that each member of the group should work 25 hours per week. As both members preferred having a flexible schedule, we did not have set working hours as long as both members worked at least the 25 hours we had agreed on every week.

5.2 Workflow

When choosing a development model for this project, it was an easy decision to go for a "Scrum Lite" model. As we are only two people, and we have worked together on multiple projects in the past, there was little need for having a lot of meetings. We decided to go for weekly sprints with an end-of-sprint meeting where we would go over the previous week's work. We would also look at what we would be doing the next week, and supplementing with additional meetings during the week if we felt this was necessary.

5.2.1 Development Workflow

When implementing a new feature, the person taking it would put their name on the related card in Trello and move it from the "Sprint Backlog" tab to the "In Progress" tab, the feature itself would be implemented in a feature branch. Whenever a feature was ready to be merged with master, a pull request would be opened and the related card would be moved to "Review" in Trello. The other group member would then have to review the code before it could be merged, and either move it to "Done (Sprint X)" if it was accepted, or back to "In Progress" if it was not. To make sure everything went through a review, we locked the master branch from accepting direct commits. Whenever either of us found a bug in the codebase, we would submit an issue on GitHub, and whoever it was more relevant to, or who had time to fix it would take the issue.

5.2.2 Performance Testing and Optimization

As we were beginning to tackle performance issues with the game we wanted a method of measuring performance. We needed this to determine if our attempts to optimize the game were successful. Without a way to measure the performance, we had no good objective way of telling if our changes to the code were actually improvements. This lead to the creation of two benchmarks; *SynBench* and *RealBench*. SynBench measures the performance of the world generation, specifically the time it takes to generate a certain area of the game world, while RealBench measures the frame-rate performance of the game. When optimizing the code we used the benchmarks to compare the game performance before and after changes made to the code. This gave us a productive and data based optimization workflow. To read more about the benchmarks and specific optimizations, see Chapter 7.

The benchmarks were also used to document performance changes in the pull requests. We did not enforce a policy of doing tests for every pull request, but it was

expected that tests would be done and documented for any pull request that could have an impact on the performance, either good or bad.

5.2.3 Bug Testing and Debugging

To find bugs in the game we would play the game while trying to break it. This means that we would not play like a normal player, but do unusual things in order to trigger bugs. This could be things such as flying into walls to see if anything breaks. We would also test for bugs by playing the game for an unusually long period of time. This is good for uncovering long term issues with the code, such as memory leaks.

As our game grew in complexity so did the bugs. The bugs could be hard to reproduce, making the process of fixing them hard. This prompted us to develop an in-game debugging tool. The debugging tool exposes key parts of the internal game state when enabled. This was an improvement over the old debugging process, where we would encounter a bug, put print statements in suspected areas of code and reproduce the bug. With the debugging tool we could debug a bug the first time we encounter it, saving us the trouble of writing print statements and reproducing the bug. See Figure 4 for an image of the debugging tool.

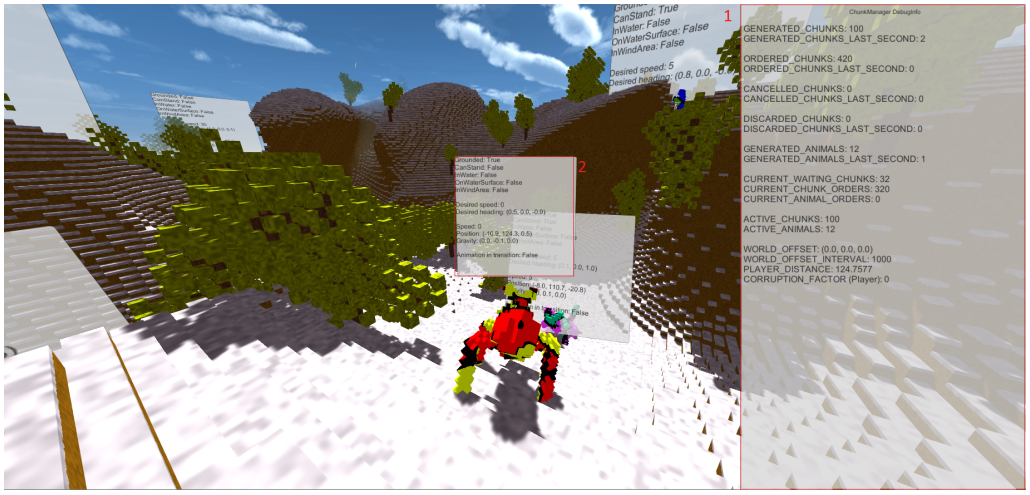


Figure 4: Debugging tool revealing internal game state. The element labeled 1 shows data for the world and world generation. The element labeled 2 shows data for a specific animal.

5.2.4 Usability Testing

Our usability testing differs from the other types of testing in that it is not carried out by us. It is hard for us to gauge the usability of the game because we developed it, and as a consequence understand how the game works. Because of this we had external play tester play the game and fill in a questionnaire with some usability questions made by us. To read more about the usability testing see Chapter 8.

5.3 Coding Conventions

Coding conventions are important for keeping a code base maintainable, so we put down some rules for how the code should look.

- Classes and enum types, and static functions should be CaptialCase.
- Non-static functions should be camelCase.
- Be explicit about accessibility levels of class members (private, public, etc).
- Enum members should be UPPERCASE, with underscores separating words.
- "this" and "base" should only be used if necessary because of context (eg. function parameter has same name as member variable).

5.3.1 Code Documentation

Functions, classes, structs and enums should be commented using XML comments, following Microsoft's recommendation [8]. If part of a function body is particularly tricky or unclear, or if a function is very large, then this should be commented as well.

5.4 Development Tools

Version Control

For Version Control we decided on using **Git**, as this is the version control system we are familiar with. We use **GitHub** for hosting our repository, as we are both familiar with using it and we both think it works well.

Project Management and Issue Tracking

We decided to use **Trello** for project management and for organizing our work, as Trello is an easy to use tool that we both have experience using. For issue tracking we used GitHub's built-in issue tracker, as this would let us easily link Pull Requests and commits to any related issues using smart commits.

Documentation

For documentation of the project we had a shared **Google Drive** folder. This contains design documents, meeting logs and any other project documentation. We used Google Drive because it allows for real time collaboration on the documents online. We also documented through our pull requests on GitHub, detailing changes made in the pull request.

Communication Tools

As we worked primarily from our homes, we needed a good way of communicating online. We ended up using **Discord** as our main communication channel. The reason for using Discord was that it is simple to set up servers and organize both text and voice channels. Discord's text chat also offers the ability to upload images and other media to share this with the other member. We had a private Discord server set up that we would use to discuss anything related to the project. We also used the built in voice chat for our weekly sprint meetings and any other meetings we had.

Game Engine

As we did not want to build an engine from scratch, we decided to go with an existing Game Engine. Our choice of game engine was the **Unity Engine**. The other alternative we considered was Unreal Engine. Our reason for choosing Unity over Unreal is that we both have a good amount of experience using Unity, and near none using Unreal Engine. Unity also has a much larger community around it, making it much easier to find information on how to make the best use use of the engine.

Coding Environment

For our IDE, we went with **Visual Studio 2017**. The Unity Plugin for Visual Studio allows us to use Visual Studio's debugging tools with Unity.

Time Tracking

For tracking time we used a tool called **Toggl**. Toggl allows us to easily categorize our work, so that we could see what our working time was spent on. Their desktop and mobile apps have real time synchronization with their servers, so that we could have access to the logs anywhere. They also offer an overview of the work done over the week, which would make it really easy to see where our time went.

Misc Tools

GIMP was used briefly early on in the project for creating terrain textures before we implemented procedural texture generation.
Audacity was used for editing audio tracks we used in the game.

6 Implementation

Our implementation focuses on the process of generating an entire 3D world populated with terrain, trees and animals. The base building block for the game world is a unit cube, known as a voxel [9], in our code and thesis we may use the words: "cube", "block" and "voxel" interchangeably when referring to voxels. We group voxels together into chunks, and then we group chunks together to get the game world. This grouping of space into voxels, and voxels into chunks gives us a world in separate manageable pieces, as opposed to having the entire world being in one piece. The practice of grouping voxels into chunks like this is used in other games that procedurally generate the world such as Minecraft [10].

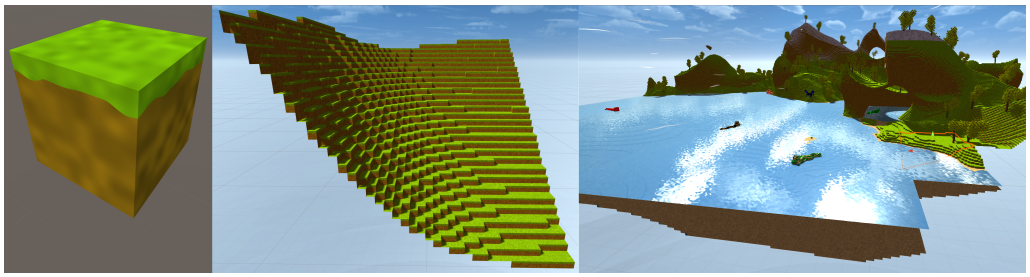


Figure 5: From left to right: A single voxel, a chunk of voxels and a world of chunks with one highlighted chunk.

We chose to generate a voxel terrain as opposed to a height map terrain [11] for its ability to represent complex 3D structures. With height map terrain you sample the height as a function of the horizontal position f(x, z) in our case (Unity uses Y as vertical coordinate). Because functions are special cases of relations, where every input is mapped to exactly one output [12], they fail to represent structures such as overhangs, which we want in our world and can be seen on the right side of Figure 5.

There are some major classes in our implementation that handle the logic of generating and maintaining the world. The main class is the `WorldGenManager`, which keeps track of every existing chunk and animal, and orders new chunks/animals from the `ChunkVoxelDataThread` threads as needed. The `ChunkVoxelDataThread` threads receive orders from the `WorldGenManager` to generate some content for the world, which it executes using the `ChunkVoxelDataGenerator` to generate terrain, `LSystemTreeGenerator` for trees or `AnimalSkeleton` to generate animals. The data generated by one of these classes is then sent to the `MeshDataGenerator` inside the thread to get the data needed for meshes. The mesh data is then sent back from the `ChunkVoxelDataThread` to the `WorldGenManager` which then deploys the generated content into the world.

6.1 Generating Data

As the basis of our world generation we need some method of generating data. We want to generate a continuous 3D world with landscape that progresses naturally from one point to another. We need the world to be consistent and deterministic, meaning that a location in the world has to be generated the same way every time it is generated. With these requirements we need a data generator that is deterministic and not entirely random meaning that closely related inputs should have related outputs.

6.1.1 Procedural Noise Functions

The most obvious way to generate data when programming is by using random number generators. Random number generators fail to meet the requirements that we have, the different numbers generated have a weak correlation which would prevent us from having terrain that progresses naturally.

A more popular way of generating data for procedural generation is by using noise functions [13]. Noise functions takes a coordinate as input and returns a noise value, usually in the 0 to 1 range or -1 to 1 range. The noise value returned by the function changes gradually as you change the input, meaning that the noise function will return similar values for input coordinates that are close to each other. Noise functions can also be deterministic, so calling the noise function with the same coordinate as input always returns the same value. Noise functions can be thought of as sampling a noise plane(in the case of 2D noise functions), these planes can be virtually infinite in size with every point containing a noise value. See Figure 6 for an image of two noise planes produced by simplex noise.

Noise functions can also be combined with two control variables, frequency and seed. The frequency controls how quickly you traverse the noise plane as you sample and the seed controls where in the noise plane you are sampling. Using a low frequency would generate fewer large scale features and using a high frequency would generate a lot of small scale features. Using noise functions with frequency and a seed would be done like this: `noise(coordinate * frequency + seed)`. See Figure 6 for a comparison of low and high frequency noise sampling.

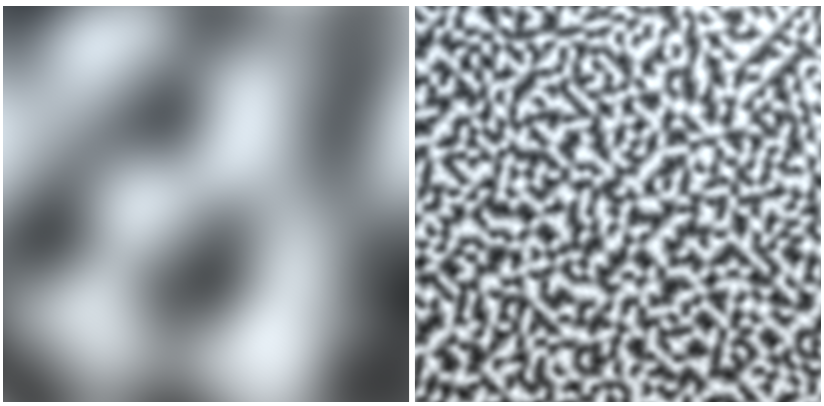


Figure 6: From left to right: low frequency noise sampling and high frequency noise sampling. The noise function was called with the pixel coordinates as input. Noise values are used as gray-scale color values to make the pictures.

Various noise functions share some core functionality, the type we are interested in is known as *lattice gradient noise* [13]. They partition space into a lattice using some primitive. They take the input coordinate and find the primitive containing the input coordinate. Then they find the corners of the primitive and use them in some calculations. Hash functions are usually used in these calculations to give them pseudo-random characteristics. Once they have calculated a noise value for every corner of the primitive containing the input they work out how much each corner should contribute to the final noise value, usually based on the distance from the corner to the sampling point.

6.1.2 Simplex Noise

Simplex noise is the successor to Perlin noise, both of which were created by Ken Perlin [14] [15]. The main difference between simplex and Perlin noise is how they partition space. Perlin noise partitions space into N dimensional cubes, whereas simplex noise partitions space into the simplest shape for the given dimension. For the first three dimensions these shapes would be: a line, a triangle and a tetrahedron. For Perlin noise the shapes are: a line, a square and a cube. The benefit of using a simplex instead of a cube is that you have to sample less points when calculating noise. The number of points in a simplex grows by $1 + n$ where n is the number of dimensions, for N dimensional cubes the number of points grows by 2^n. This means that simplex noise has a lower time complexity then Perlin noise. The use of simplexes was also found to decrease the prevalence of directional artifacts in the generated noise [16]. For these reasons we decided to use simplex noise instead of Perlin noise in our game.

Our implementation of simplex noise covers 1D, 2D and 3D versions of simplex noise and is based on an article written by Jasper Flick [17]. We will be explaining the 2D implementation of simplex noise here. The 2D simplex is a triangle, because of that we will be partitioning space into equilateral triangles. We will break the noise sampling down into 4 steps which are:

1. Finding the corners of the equilateral triangle containing our point.
2. Calculating the noise values for each corner.
3. Calculating the falloff for each corner.
4. Combining the corners into a final noise value.

Finding the corners of the equilateral triangle containing our point

The coordinate system already gives us a space partitioned into squares. For any given point you can floor its position and add 1 to its components to find the 4 corners of the square containing it as seen in Figure 7. If we split the square along its diagonal we get two right isosceles triangles as also seen in Figure 7. When we sample noise we want to find the corners of the equilateral triangle containing the point we are sampling. There is no obvious way of finding the corners of the containing triangle as there is with the containing square mentioned earlier. For this reason we want a method of transforming our equilateral triangle into one of the triangles shown in Figure 7. Finding the corners of the transformed triangle is easy and by transforming them back into an equilateral triangle we get the corners we want.

We now need some method of doing the actual transformations between the equilateral triangle grid and the right isosceles triangle grid. For an image of the two grids see Figure 8. We will start by working out how to transform a right isosceles triangle

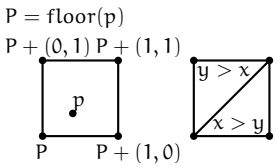


Figure 7: A point inside a square and the corners of the square. A square split in half.

into an equilateral triangle. We need to scale every point along the main diagonal. This can be done by subtracting how far along the diagonal the point is multiplied by some factor S. This scaling factor that we subtract by is: $S(x+y)$ and it is used like this: $P = (x - S(x+y), y - S(x+y))$.

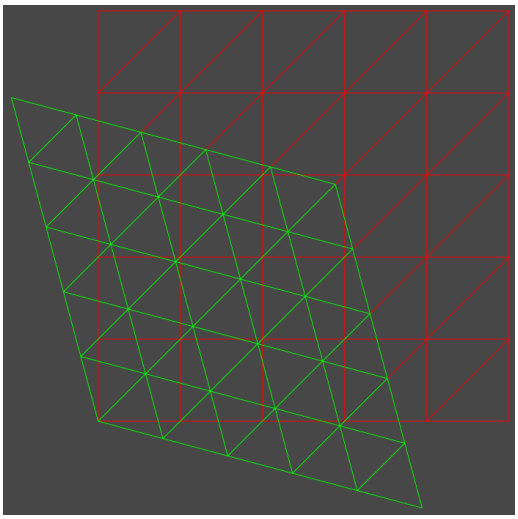


Figure 8: The equilateral triangle grid was produced by transforming the right isosceles triangles using $S(x+y)$.

To work out the scaling factor S in $S(x+y)$ we can consider the case of transforming the triangle ABC into its equilateral version $A'B'C'$. ABC is defined by $A = (0,0)$, $B = (0,1)$ and $C = (1,1)$. We can make a quadratic equation to find S since we know that the resulting triangle has sides of equal length. We choose $|A'B'| = |A'C'|$, since A is zero we can ignore it and simply do $|B'| = |C'|$. B' is given as $B' = (-S, 1-S)$ and C' is given as $C' = (1-2S, 1-2S)$. C' and B' were obtained by subtracting the scaling factor $S(x+y)$. Calculating S for transforming into the equilateral triangle grid can be now be done as follows:

`We will be using the squared lengths to avoid square roots.`
$|B'|^2 = |C'|^2$
$S^2 + (1-S)^2 = 2(1-2S)^2$
$2S^2 - 2S + 1 = 8S^2 - 8S + 2$
$-6S^2 + 6S - 1 = 0$
`Solving the quadratic equation gives the solutions:`
$S = (3-\sqrt{3})/6$ `and` $S = (3+\sqrt{3})/6$.

We want the smaller of the two solutions as our S in $S(x+y)$ because the other solution would create negative triangles. We now have $S = (3-\sqrt{3})/6$ for transforming into the

equilateral triangle grid. We now need a second S for transforming out of the equilateral triangle grid. The second S can be found by reversing the the transformation we just did. The point $C' = (1-2S, 1-2S)$ was transformed to $C' = (1-2(3-\sqrt{3})/6, 1-2(3-\sqrt{3})/6)$ and it has to become $C = (1,1)$ to transform out of the equilateral triangle grid. By solving it for $C'.x = C.x$ we can find the new S. C' now has to be transformed out of the equilateral triangle grid so this time we add by the scaling factor $S(x+y)$:

Start by simplifying $C'.x$
$C'.x = 1 - 2(3-\sqrt{3})/6 = 1/\sqrt{3}$
Now we add the scaling factor to $C'.x$.
$x = y$ in this case so $S(x+y)$ becomes $2Sx$ giving
$S(x+y) = 2S/\sqrt{3}$
$C'.x$ then becomes $C'.x = 1/\sqrt{3} + 2S/\sqrt{3}$
This give us the equation:
$1/\sqrt{3} + 2S/\sqrt{3} = 1$ for $C'.x = C.x$
Isolating S gives:
$S = (\sqrt{3}-1)/2$

We now have our two scaling factors $(3-\sqrt{3})/6(x+y)$ for transforming into the equilateral triangle grid and $(\sqrt{3}-1)/2(x+y)$ for transforming out of the equilateral triangle grid. We can now find the corners of the equilateral triangle containing our point. We start by transforming our point P out of the equilateral triangle grid. We then floor P giving us the two corners along the diagonal: $\mathtt{floor}(P)$ and $\mathtt{floor}(P) + (1,1)$. To find the last corner not on the diagonal, we need to work out which of the two triangles making up the square we are in, as seen in Figure 7. We look at the fractions of x and y, the fractions gives us our local coordinate for the square. The comparisons in Figure 7 is then used to determine if the last corner is $\mathtt{floor}(P) + (0,1)$ or $\mathtt{floor}(P) + (1,0)$. Now that we have our corners we can transform all of them back to the equilateral triangle grid and use them for calculating noise.

Calculating the noise values for each corner

Simplex noise is a type of noise function known as *gradient noise*. This means that for every point we sample we map that point to a gradient which we use for calculating the noise. Simplex noise uses a hash to map a point to a gradient, giving the noise pseudo-random characteristics. The hashing uses a `int[] hash` array with the numbers 1-255 in a random order combined with `int hashMask = 255` and bit-wise logic on the point in question. The hash value for a corner is calculated as: `int hashValue = hash[hash[ix & hashMask] + iy & hashMask]` where `ix` and `iy` is the x and y of the corner. The gradients are implemented in a similar manner to the hash with a `Vector2[] gradients2D` array containing 8 evenly spread directional vectors and a mask `int gradientsMask2D = 7`. By taking the calculated hash value for the corner we find the gradient for the corner using `Vector2 gradient = gradients2D[hashValue & gradientsMask2D];`.

Once we have the gradient for the corner we calculate the noise value as: `float cornerNoise = dot(gradient, (point - corner))` where point is the point that the noise function was called with.

Calculating the falloff for each corner

We need to compute a falloff value for every corner as well that we multiply the corner noise value by. This makes it so that corners close to the sampling point contributes more then distant points for the final noise value. We want a falloff function that starts at 1 when the distance $D = length(point - corner)$ is 0. This makes it so that a corner would contribute fully to the noise value when it equals the sampling point. We also want the falloff function to reach 0 when the distance D reaches the height of the equilateral triangle and stay at 0 for distances longer then that. We can start with $(1 - D^2)^3$ as our falloff function and adjust it for the height of the triangle. $(1 - D^2)^3$ is 1 when D is 0, its value is 0 when $D = 1$ and the derivative is zero for $D = 1$.

We now have to adjust 1 in $(1 - D^2)^3$ for the height of the equilateral triangle and we are almost done. The height of an equilateral triangle is calculated by taking the length of an edge and multiplying it by $\sqrt{3}/2$. We can use the $A'B'C'$ triangle from Chapter 6.1.2 to calculate the edge length of our equilateral triangles.

We can use $|B'|$ from $A'B'C'$ as our edge.
With $B' = (-S, 1 - S)$ and $S = (3 - \sqrt{3})/6$ we get:
$|B'|^2 = 2/3$
$|B'| = \sqrt{2}/\sqrt{3}$
The height then becomes:
$H = (\sqrt{2}\sqrt{3})/(2\sqrt{3})$
$H = \sqrt{2}/2$

The height of our triangle is $\sqrt{2}/2$, we are using the squared distance in our falloff function, so we should square the height also. This gives us the falloff function: $(1/2 - D^2)^3$. Our falloff now starts at $(1/2)^3$ so we have to scale the result by the end to get correct values.

Combining the corners into a final noise value

We compute the noise value for every corner multiplied by the falloff, then we add them together an normalize the result. The maximum value occurs when the sample point is at the center of the triangle, with all gradients pointing towards the center. The distance from a corner to the center is calculated by taking the edge length multiplied by $\sqrt{1/3}$. This gives us the distance $D = \sqrt{2}/3$ (using the edge length calculated earlier). The noise value for a corner is calculated as `float cornerNoise = dot(gradient, (point - corner))`, in this scenario the point is at the center and the gradients are pointing at the center. This makes `gradient` and `(point - corner)` parallel and the dot product can be computed as $|gradient||point - corner|$, since the gradient is a unit vector it can be reduced to $|point - corner|$. The point being at the center gives: $|point - corner| = \sqrt{2}/3 = D$. The noise value of one corner then becomes $D(1/2 - D^2)^3$ which is the noise value multiplied by the falloff, in this case the noise value equals the distance also. There are three corners so the final maximum un-scaled noise value is:

$3D(1/2 - D^2)^3$
Substituting D with $\sqrt{2}/3$ yields the maximum value:
$125\sqrt{2}/5832$
Inverting and simplifying yields the scaling factor:
$2916\sqrt{2}/125$

We can now take the inverse of the maximum noise value $2916\sqrt{2}/125$ and multiply it by the sum of the noise from the corners to produce our final noise value for the point inside the equilateral triangle.

Scaling up to 3D noise

There are a lot of similarities between 2D and 3D simplex noise. For 3D noise space gets partitioned into tetrahedrons instead of equilateral triangles. And we transform the points between the tetrahedron grid and cube grid for the same reasons as in 2D. The logic obviously gets a lot more involved in 3D, tiling space into equilateral triangles is easier then doing it with tetrahedrons. However the core principles still applies.

6.1.3 GPU Noise

We use noise on the GPU as well for generating textures for the terrain and animals, see Chapter 6.9. It is important to us that the noise function used on the GPU is fast, because it could be called once for every pixel on the monitor or more. Because of this we have not used simplex noise, but a simpler implementation of a 3D *lattice gradient noise* function. The noise function we use is based on HLSL port [18] of a noise function made by Inigo Quilez [19]. It samples noise from the integer lattice grid, calculating a noise value for each point using a hash function, then the points gets interpolated into a final noise value. The function is almost like Perlin noise [14] without the gradients.

Listing 6.1: GPU noise hash function

```
float hash (float n){
        return frac(abs(sin(n) * 43758.5453));
}
```

Since the noise function is sampling points from the integer lattice grid, the points of the bounding integer cube for our sampling point can be found with the method from Figure 7. The noise value for one of the corners of the integer cube is the hash value calculated by the noise function in Listing 6.1, the `abs` term makes the final noise value in the 0-1 range. Since the hash function takes a scalar input we convert the sampling points to a scalar using: $scalar = point.x + point.y * 57 + point.z * 113$ before calling the hash function. Once we have a noise value for every corner we combine them into a final noise value using interpolation. The interpolation factor used is $t = frac(inputPoint)$, we then calculate a falloff for t as such: $t = t^2(3 - 2t)$. We start by interpolating all of the 8 noise values along the X axis, in pairs of 2 using t.x. This halves the initial 8 values to 4, we repeat the process in the Y axis giving us 2 points, then finally do the process one more time in the Z axis giving us the final noise value. See Figure 9 for an image of the interpolation process applied to 4 points.

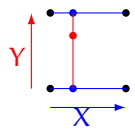


Figure 9: The black dots are the initial points that noise values have been calculated for. The noise values are interpolated along the X axis as shown by the blue lines and dots. Then interpolated along the Y axis as shown by the red line and dot. The interpolation factor used is the fractions of the input point.

6.1.4 Poisson Disk Sampler

Poisson Disk Sampling is a non-uniform sampling pattern that will spread points in a uniform way so that points will not cluster together and wont leave large empty spaces, as opposed to random sampling where the points will cluster together and there will be large empty spaces, which can be seen in Figure 10. Our approach to the Poisson Disk Sampler is based on an article by Robert Bridson [20].

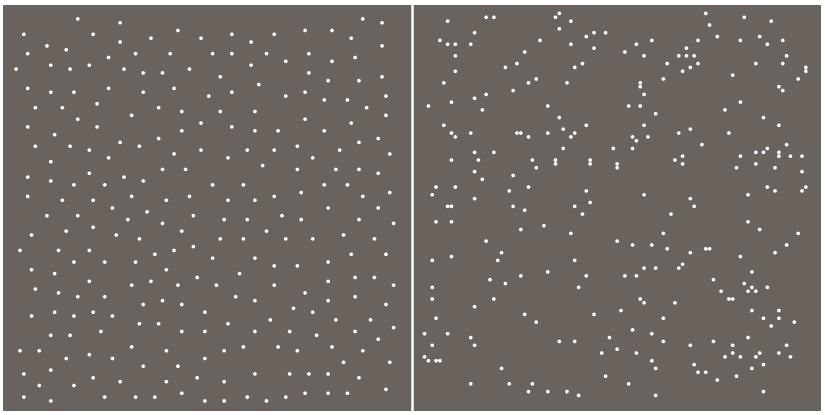


Figure 10: Comparison between Poisson Disk sampled points and randomly sampled points. Left: Poisson Disk Sampling, Right: Sampled using uniform distribution.

Before we start sampling we create a list `activePoints` that contains all the points that we can still sample for neighbours, and a 2 dimensional boolean grid `doneGrid` of where points have been placed covering the sampling domain. We choose a random location in the sampling domain, check it off in the grid and add it to `activePoints`. For as long as there is at least one point in `activePoints`, we take a random point out of the list for sampling. Once the list becomes empty, we are done sampling.

Sampling a point

When sampling a point, we generate up to 30 points in the annulus of the sampling point. The annulus is between **R** and **2*R** away from the sampling point, where R is a set radius. For each of these points, we validate that they are not to close to any existing point on our grid. When we find a point that is valid, we add it to `doneGrid` and to `activePoints`, and consider ourselves done with this sampling point for now. If we manage to generate all 30 points without finding one that is valid, we remove the sample point from `activePoints`. When we're done with the sampling point we choose a new random point from `activePoints` to check next.

Point validation

To validate a point we check the points in `doneGrid` for any points within **R** of the point we are checking, if we find any points within **R** of the point, then the point is not valid. Figure 6.1.4 shows an example of a valid test and an invalid test against an existing point. The figure obviously only shows one existing point, but the same test is run against all existing points.

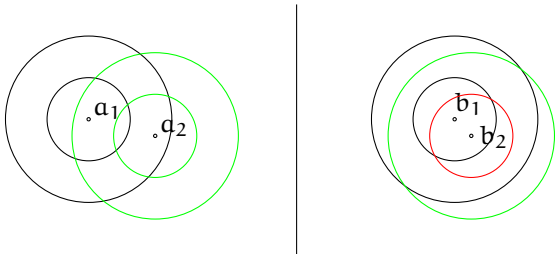


Figure 11: Two points being checked against existing points. Left: Point a_2 is tested against existing point a_1. Right: Point b_2 is being tested against existing point b_1, but is to close.

6.2 Procedural Generation

The core of our procedural generation is the `BlockDataMap`. Whenever we generate something in the world, the data for that something is stored in a `BlockDataMap`. That BlockDataMap is then sent to the `MeshDataGenerator` to generate a mesh to place in the world. The `BlockDataMap` contains an array of `BlockData` objects, the array is one dimensional but it represents 3 dimensions and we index it with x, y, z. `BlockData` contains the block type information on a voxel, one for the base type of the block (Water, Dirt, Sand, Wood, Leaf or Animal Skin) and a modifier type (Grass or Snow) which is used for generating textures (Chapter 6.9) and for audio (Chapter 6.8).

6.2.1 Mesh Generation

We have two methods for generating mesh data, one is a `NaiveMeshDataGenerator` which simply generates a face for every side of a block where it does not have a neighbour, and the other is a `GreedyMeshDataGenerator` which optimizes the number of faces it creates to minimize the vertex count. The reason for having two `MeshData` generators is that we need the `NaiveMeshDataGenerator` for animals as they need more vertices for animation, while the `GreedyMeshDataGenerator` is needed for the terrain as we want to optimize the mesh as much as possible.

The `MeshData` generators don't generate the actual mesh, only the data needed to create the mesh. This is because the `MeshData` is generated in a worker thread, and Unity does not allow for use of the Mesh class outside of the main thread. How this and the rest of our multithreading system works is explained in detail in Chapter 6.4.1.

Naive Mesh Generation

To generate mesh data, the `NaiveMeshDataGenerator` goes through every block in the `BlockDataMap`, and for all six sides it checks the neighbouring block on that side to see if that block is solid or not. If the neighbouring block is not solid, a face will be generated, but if the block is solid no face is generated, as the face would never be visible, and would have no use for collision. The `NaiveMeshDataGenerator` takes in a `MeshDataType`, which helps discern how the mesh should be generated. If the `MeshDataType` is water, then it will only generate faces for water blocks and if it is terrain it will treat water blocks like as if there was no block there. Whenever the generator is generating `MeshData` for water, the neighbour check does not check if the neighbour is solid, but if it is also water. Faces

are then generated whenever the neighbour is not water.

To generate a face (consisting of two triangles) for a block, the `NaiveMeshDataGenerator` uses the direction and the center point for the block to calculate the vertex positions, normal values and texture coordinates. If the `MeshDataType` is `MeshDataType.TERRAIN` or `MeshDataType.TREE`, it will embed the `BlockType` data in the vertex color so that textures can be generated on the GPU. If the `MeshDataType` is `MeshDataType.ANIMAL`, then we embed a seed in the UV for generating the texture on the GPU. How the textures are generated is explained in Chapter 6.9.

Greedy Mesh Generation

The `NaiveMeshDataGenerator` was fairly simple to implement, but it is not very optimal. While it does cull faces that will never be displayed, if you had a mesh made from 10x10x1 blocks, there would be 240 faces, or 960 vertices needed for this mesh, while an optimal solution would only need 6 faces, or 24 vertices. As the number of blocks increase the difference between the naively generated mesh and the optimal mesh becomes even larger, so we wanted to find a way to optimize the vertex count because it was affecting the frame rate in the game. This is what lead to the implementation of the greedy mesh generator. In the end, we were not actually able to use the `GreedyMeshDataGenerator` for the terrain because it results in visual artifacting as a result of T-junctions causing floating point imprecisions [21].

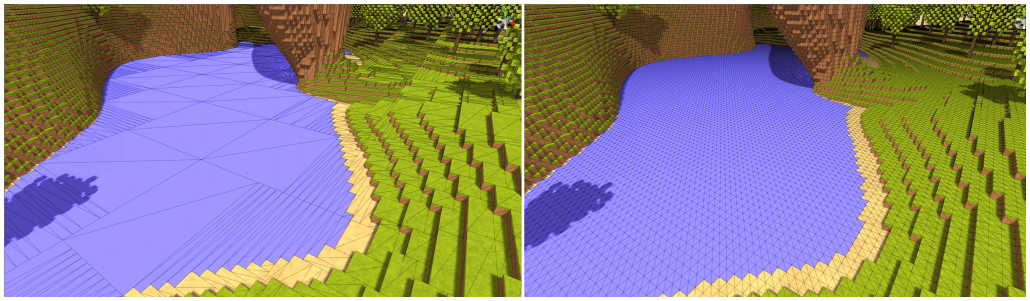


Figure 12: The effect of the optimized mesh can be seen very clearly, especially on the water. (Left: Greedy, Right: Naive)

The `GreedyMeshDataGenerator` is based on an article and an implementation by Mikola Lysenko [22], the difference between our implementation and his implementation is that ours support multiple block types, while his version is a binary voxel or no voxel.

The way the `GreedyMeshDataGenerator` works is by looking at the problem as a series of 2D grids instead of as a 3D grid. When going through the `BlockDataMap` we take one layer at a time and create a mask that can be used for generating the `MeshData` itself. The mask is made up of a 2D array of `VoxelFaces`, where each `VoxelFace` contains `BlockData` and whether the face is flipped.

To create the layer mask we go though each point in the layer and look at what type of block it is, we also look at the neighbouring same point in the next layer down. If both points contain a block, or neither contains a block, nothing is added to the layer mask. If only one of the points contain a block we add a new `VoxelFace` with the `BlockData` of the block. If the point with a block was in the next over layer, we set the `VoxelFace` to be

flipped. After creating a layer mask, we can generate the actual mesh data for this layer.

To generate `MeshData` from the layer mask we go through the mask starting at (i, j). We then calculate the width by going through the x-direction until we reach the end of the layer mask, or we find a different `VoxelFace`. The width is then stored in a variable w. When we have the width, we can calculate the height. To calculate the height we go through the y-direction and check every `VoxelFace` from x=i to x=i+w in until we find a `VoxelFace` that does not match the original one. Once we find a non-matching `VoxelFace`, we get the number of full rows and store that as the height in a variable h. We now have a quad (i, j, i+w, j+h) which we can use together with the `VoxelFace` to generate mesh data. The i is then incremented by w and we can start again at the new (i, j) and keep going until we reach the end of the layer mask.

To generate the `MeshData`, we use the included quad to generate the vertices, and the members of the `VoxelFace` to generate normals and texture coordinates. Just like in the `NaiveMeshDataGenerator` we also embed the `BlockType` data for texture generation on the GPU if the `MeshDataType` is `MeshDataType.TERRAIN` or `MeshDataType.TREE`. The texture generation is explained in Chapter 6.9.

The process of going through the layers, creating a layer mask and generating the mesh data from the mask is repeated for all 3 dimensions.

6.2.2 Terrain Generation

There are 3 phases to generating the terrain of a chunk. In the first phase we generate three 2 dimensional maps covering the chunk, these are the biome map, the corruption map and the heightmap. In the second phase we generate the 3D block map with 3D noise through selective sampling. The final phase is going over the terrain and setting the actual types for each block in the terrain and adding water where that goes.

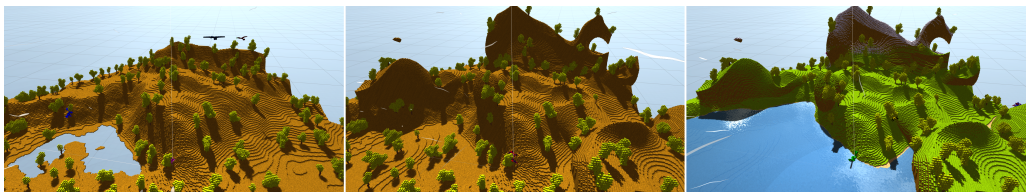


Figure 13: The 3 phases of terrain generation. From left to right: 2D terrain only, 3D terrain without block types and finished 3D terrain.

Phase 1: Generating the 2 dimensional maps

We start off by generating the biome map and the corruption map for the chunk. The biome map is a 2D array containing lists of pairs of biomes and their weight on the position. How we find the biomes and their weights for a position in the map is explained in detail in Chapter 6.2.3. The corruption map contains the corruption factor for the position. Corruption is explained in detail in Chapter 6.2.4. The final 2 dimensional map we have to generate for the chunk is the heightmap, this contains the 2 dimensional noise used to generate the terrain.

The heightmap is generated using multiple octaves, this is so that we can have different levels of features, ranging from mountains to small mounds. We have an `octaveStrength` which starts at 1, but halves with every octave. This `octaveStrength` is used to control

the frequency and amplitude of the noise we are sampling. For each octave, we calculate a weighted average of a 2D simplex noise sample from all biomes in `biomemap[x,z]`. The noise sample we create for each biome uses the frequency from the biome divided by the `octaveStrength`. Once we have calculated the weighted average we add the resulting value multiplied by the `octaveStrength` to a variable `finalNoise` and add the current `octaveStrength` to a variable `noiseScaler`. When we have gone through all the octaves, we normalize the noise value by dividing the `finalNoise` by the `noiseScaler`.

Each biome can contain a minimum and a maximum ground height, so we calculate the weighted average of both minimum and maximum ground levels for all biomes in `biomemap[x,z]`. We can now set `heightmap[x,z] = minGroundHeight + finalNoise * (maxGroundHeight - minGroundHeight)` so that the value in `heightmap[x,z]` is how many voxels there are between the ground and the bottom of the world.

After we have generated the biome map, corruption map and heightmap we initialize the `BlockDataMap` for the chunk with the data from the heightmap. The reason for this is that we do not sample every block in the `BlockDataMap` in phase 2 for performance reasons, so initializing everything here makes sure that all blocks have been initialized.

Generating the 3 dimensional block map

As sampling 3d noise for every block in the `BlockDataMap` would be to expensive (See Chapter 7.3), we place any blocks we want to sample into a sampling queue. We also have a 3D boolean array where we check off positions once they have been added to the queue, so that we don't end up sampling a block more than once. The queue is initialized by adding `(x, heightmap[x,z], z)` for every x and z in the chunk, and all the positions of at the sides of the chunk. When we pop a position off the queue, we sample this position with a combination of 2D and 3D noise, the block at the sampled position in the `BlockDataMap` is then updated with the result from the sampling. If the block ended up changing from `BlockType.NONE` to `BlockType.DIRT` or vice versa, we queue up any neighbouring positions that have not yet been queued, and is within the bounds of the chunk. This process is repeated for as long as there are positions in the queue to sample.

To decide whether a position should contain a block or not, we generate three 3D noise values that we combine with the already generated 2D noise. The first of the 3D noise values is the `structure` noise, this is used to decide where in the world (in addition to the terrain generated by the heightmap) we want to have blocks. To calculate the `structure` noise, we use 3D simplex noise (see Chapter 6.1.2) which is interpolated down towards 0 the closer the sampled position's y-value is to the world height, so that we don't get any flat cutoffs at the top of the world. The second of the 3D noise values is the `unstructure` noise, which is used to decide where we in the world we do not want to have blocks. `Unstructure` noise, like the `structure` noise uses 3D simplex noise, but interpolates up towards 1 the closer to 0 our sampled position's y-value is instead, so that we don't get any holes in the ground. The final noise value we need is the `corruption`. This value is, just like the previous two values using 3D simplex noise, the `corruption` is explained further in Chapter 6.2.4. If the biome map generated earlier contains more than one biome in `biomemap[samplePos.x,samplePos.z]`, then these 3 noise values are calculated for every biome and a weighted average is calculated based on the weight of each biome in `biomemap[samplePos.x,samplePos.z]` (see Chapter 6.2.3).

In addition to the sample values we calculated for `structure`, `unstructure` and

corruption there are also cutoff points for each of them stored in the biomes. Just like with the noise values we use weighted averages if there is more than one biome in biomemap[x,z]. There is also the corruptionFactor, which increases the further from (0,0) the sample position is and is used for interpolating the corruption cutoff down towards 0 the closer to (0,0) the sample position is. Together with the 2D height value from heightmap[x,z] we send these value to a function posContainsVoxel() to check if there should be a block at the sampled position.

Listing 6.2: posContainsVoxel() function.

```
bool posContainsVoxel(Vector3 pos, int height,
    float structure3DRate, float unstructure3DRate, float corruptionRate,
    float structure, float unstructure, float corruption, float corruptionFactor) {

    bool inHeight = pos.y < height;
    bool inStructure = structure3DRate > structure;
    bool inUnstructure = unstructure3DRate < unstructure;
    bool inCorruption = corruptionRate * corruptionFactor > corruption;
    return (inHeight || inStructure || inCorruption) && (inUnstructure);
}
```

Finalizing the block map

Now that we have generated the block map, we can go through all the blocks and finalize their block types. If the blocks type is BlockType.DIRT then the block types are decided by the biome(s) in which a block is. If there is only one biome, then that biome simply decides the block type, but if there are multiple biomes the biome used is selected randomly, with the biomes having a non-equal chance of being chosen based on a calculated weight. This weight is calculated as weight=Mathf.Pow(oldWeight*0.25f,2), where oldWeight is the weight already stored with the biome in biomemap[x,z]. The reason we're not using the weight in the biome manager directly is that the transition wouldn't look good, so we tweak it to make the transition look better. After setting the block type for all the blocks, we have to add the snowline. For this we find the weighted average of the snow lines from the biomes the block is on, and if the block is above the snow line we set the blocks modifier to BlockType.SNOW. See Chapter 6.2.3 for how the biome does its part in deciding the block types.

While going through and setting the block types for all the blocks, we also place out water. Whenever we are at an block whose block type is BlockType.NONE we check if it is below the water level. If the block is below water level we add in a water block at (x, y + corruptionWaterHeight, z), where corruptionWaterHeight is calculated using the corruption map from phase 1, how this is calculated and why we have this is explained in Chapter 6.2.4.

6.2.3 Biomes

The biome system was created to give more variety to the terrain, with areas that can vary wildly in how they look. Before we implemented the biomes, the terrain wouldn't be very different at different places in the world, but now that we have biomes, we can find everything from snowy mountains, to large oceans, to dry empty deserts. The biomes contain the values necessary for generating the terrain; noise frequencies and cutoff points for both 2D and 3D noise that is used to generate the terrain (see Chapter 6.2.2), as well as a snow line and tree density. The biomes also contain a function

`getBlockType()` used for deciding the block type of a block. All biomes inherit from a BaseBiome class, which contains the default `getBlockType()` function.

Biomes are managed by the BiomeManager. The BiomeManger stores the biomes placed in the world in a list `List<Pair<BiomeBase, Vector2>> biomePoints`, containing the biomes and their positions in the world. It is easily extensible, so new biomes can be added very easily.

We have implemented 5 biomes:

1. **Basic Biome:** This was the first biome we added, the values it uses for generating terrain and foliage are the same as the ones we used before adding the biomes. It is a generally hilly biome, with some snow covered tops and water filled valleys.
2. **Mountain Biome:** The mountain biome is mostly covered in snow, due to having a lower snow line than other biomes, as well having a much higher minimum ground level than the other biomes.
3. **Forest Biome:** This biome is a relatively flat biome, and is for the most part covered by a thick forest.
4. **Desert Biome:** Entirely covered in sand, this biome is the only implemented biome to use a custom function for getting the type of a block.
5. **Ocean Biome:** This biome is almost entirely covered in water, except for small islands that can be encountered every once in a while.

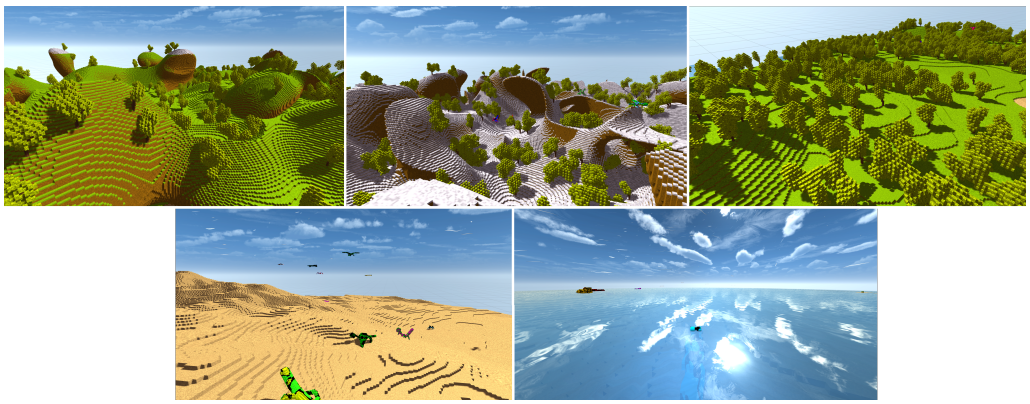


Figure 14: Biomes. From top left: Basic Biome, Mountain Biome, Forest Biome, Desert Biome, Ocean Biome.

Adding biomes to the world

The biomes are placed in the world using Poisson Disk Sampling (see Chapter 6.1.4). The reason for using Poisson Disk Sampling is that we wanted the biomes to be of comparable size, instead of having some biomes spanning very large areas, and some being very small. Poisson Disk Sampling allows for this, and generates large amounts of points in a very short time. For using the Poisson disk sampler, we generated it with a sample domain of 500x500, and a radius of 5. The points returned from the Poisson disk sampler are scaled up by a factor of 100, so that the points span a 50000x50000 block area. The world only goes out about 20000 meters from spawn before it ends, so 50000x50000 meters of biome coverage is enough. For every point from the Poisson disk sampler, we choose a random biome and add the biome and point to the `biomePoints` list. The

randomly chosen biome is also placed in a 2 dimensional array `biomeGrid` covering the sample domain, at the position given by the Poisson disk sampler.

Finding biomes in range of a point

Whenever a point (x,z) in the world is checked for in-range biomes, the first thing we do is figure out how far away the point is to its closest biome and store it in `closestBiomeDist`. Knowing the distance from the closest biome, we can find all biomes within range. The range is set to be `closestBiomeDist + biomeBorderWidth`, where the biome border width is the size of the transition area between biomes. While going through the biomes to find the ones that are in range, we also save the distance from our point to the biome in a pair with the biome itself in a list `List<Pair<BiomeBase, float>> inRangeBiomes`. The float part of the pairs will later be replaced with the weight of the biomes. If there is only one biome within range of our point, then we set the weight of it as 1 and return the list. However, if there is more than one in-range biome, we need to calculate the weights of each of these biomes. We calculate the weight of one biome using a falloff function `p.second = Mathf.Pow(1 - (p.second - closestBiomeDist) / biomeBorderWidth, 2);` where `p.second` is the float part of the biomes pair in the `inRangeBiomes` list. The reason for calculating the weight like this instead of using a linear falloff function is that the linear falloff looks bad, and through experimentation we found that this falloff function looked good. After calculating the weights for all the biomes, we normalize the weights so the sum of all the weights are 1.

Deciding the type of a block

Most biomes stick to the standard `getBlockType()` function that can be seen in Listing 6.3, with the exception of the desert biome. The function takes the BlockDataMap we want to alter, and the position of the block, the reason why the BlockDataMap is included is because the type of the block might depend on the blocks around it.

The `BiomeBase.getBlockType()` function is fairly straight forward. As the blocks are always `BlockType.DIRT` when first generated (see Chapter 6.2.2), we can assume that any block sent to this function will be `BlockType.DIRT`. The first thing we do is set the base blocktype. If the position is below the water level, it is set to `BlockType.SAND`. When the base block type has been set, we set the modifier of any dirt block with no block above it to `BlockType.GRASS`.

Listing 6.3: BiomeBase.getBlockType()

```
public virtual void getBlockType(BlockDataMap data, Vector3Int pos) {
    int pos1d = data.index1D(pos.x, pos.y, pos.z);
    int above1d = data.index1D(pos.x, pos.y + 1, pos.z);
    // Add block type:
    if (WorldGenConfig.positionInWater(pos))
        data.mapdata[pos1d].blockType = BlockType.SAND;

    // Add modifier:
    if (pos.y == WorldGenConfig.chunkHeight - 1 ||
        data.mapdata[above1d].blockType == BlockType.NONE) {
        if (data.mapdata[pos1d].blockType == BlockType.DIRT) {
            data.mapdata[pos1d].modifier = BlockType.GRASS;

        }
    }
}
```

`DesertBiome.getBlockType()` is even simpler than `BiomeBase.getBlockType()`, as

it sets all the dirt blocks to `BlockType.SAND`.

Listing 6.4: DesertBiome.getBlockType()

```
public override void getBlockType(BlockDataMap data, Vector3Int pos) {
    int pos1d = data.index1D(pos.x, pos.y, pos.z);
    // Add block type:
    data.mapdata[pos1d)].blockType = BlockType.SAND;
}
```

6.2.4 Corruption

One of the features that we wanted for the game from the beginning was corruption of the world as you progress through it. What we mean by corruption is that the world should look increasingly surreal as you move away from the origin. This includes having the world generate terrain that breaks intuition and the laws of physics. We made the corruption of the world express itself in 4 ways. These are: water elevation, 3D structures in the terrain, sky color change and sun manipulation. The degree of corruption is given by calculating a `corruptionFactor` in the 0 to 1 range, 0 being no corruption and 1 being full corruption. The `corruptionFactor` increases linearly with distance from origin, making the corruption of the world gradual. For an image of the effects of corruption see Figure 15.

Figure 15: This image shows the same area of the world with no and full corruption applied. The left image has no corruption while the right image has full corruption.

Corruption factor

The `corruptionFactor` is a value that is calculated for every horizontal voxel coordinate. It is only influenced by horizontal position (x and z coordinates). The value is calculated as the input coordinate's distance from origin divided by the maximum distance for the world, the value is then clamped to the 0 to 1 range. This creates a linear corruption of the world from start to finish. However this prevents the player from ever experiencing the non-corrupted world, so we add a grace period of a certain distance to the calculation. The grace distance is subtracted from both distances in the division, so that the `corruptionFactor` does not increase before after the grace distance. The world also ends when the `corruptionFactor` becomes 1, which is expressed as the absence of any terrain. See Figure 35 for an image of the end of the world.

Water elevation

The water elevation is implemented in such a way that it looks like the water bodies have floated out of the water bed. This means that the floating water bodies maintain

the shape they would have had if they were placed correctly in the water bed. To elevate the water we take the `corruptionFactor` of any given water block and use it to calculate a height offset for the block. The offset is calculated as `maxOffset * corruptionFactor` which is then added to the water blocks original height.

3D structures

The 3D structures works the same way as the 3D Structure noise(see Chapter 6.2.2). It is used to generate 3D structures in the world by sampling 3D noise and comparing the noise values to some cutoff. If the noise value at a coordinate is less then the cutoff value that coordinate should contain a voxel. To make the generation of the corruption 3D structures gradual we multiply the `corruptionFactor` by the cutoff so that it gradually increases in value. The corruption 3D noise is generally used with a higher frequency and a more permissive cutoff then the normal terrain noise. This is because we want it to create more pronounced features in the terrain, such as a floating ball of voxels. The normal 3D noise used for terrain used with a low frequency creates larger scale structures such as overhangs.

Sky color

The sky color change is done by interpolating between two cubemap textures used for the skybox. The interpolation factor used is the `corruptionFactor` of the player position. This makes it so that the sky slowly goes from the bright sky seen in Figure 15 to the dark sky seen in the same figure.

Sun manipulation

Since our sun is a directional light we can simulate sunsets and sunrises by rotating the light. When the `corruptionFactor` of the player goes above 0 the sun will start rotating towards randomly selected rotations between a sun set and mid day. The time of a rotation is constant, not depending on the number of degrees required to complete. This makes the various sun rotations erratic without apparent patterns, building up the theme of chaotic corruption that we want. As the `corruptionFactor` increases the speed of the sun rotations also increase.

Other implementations we considered

The above implementation of corruption was one of 4 implementations we proposed before we begun implementing.

One of the other approaches involved making a corrupt version of every biome(see Chapter 6.2.3), then we would use the `corruptionFactor` to interpolate between the two versions. This approach had the benefit of giving us a lot of control regarding how the corruption would look, since we could design the corruption for every biome. The big drawback of this approach is the fact that it would double the amount of noise sampling we would have to do, which is why it was not chosen.

Another approach was a modification of the above approach, where we would make biomes with various stages of corruption. We would then use the corruption factor to decide which version of the biome to use. This fixes the problem with the above implementation but creates two new problems. We would have to design a lot of biomes and the corruption would be introduced in various steps, reducing the granularity of its progression.

The last corruption implementation we considered but did not choose is based on using the `corruptionFactor` to influence the normal noise sampling. This could mean having the corruption influence the frequency of the noise sampling used by the normal terrain generation. The benefit of this approach is that it would be influenced by corruption gradually without requiring any extra noise sampling. The drawback was the fact that it caused severe directional artifacts in the terrain. See Figure 16 for an image of these artifacts.

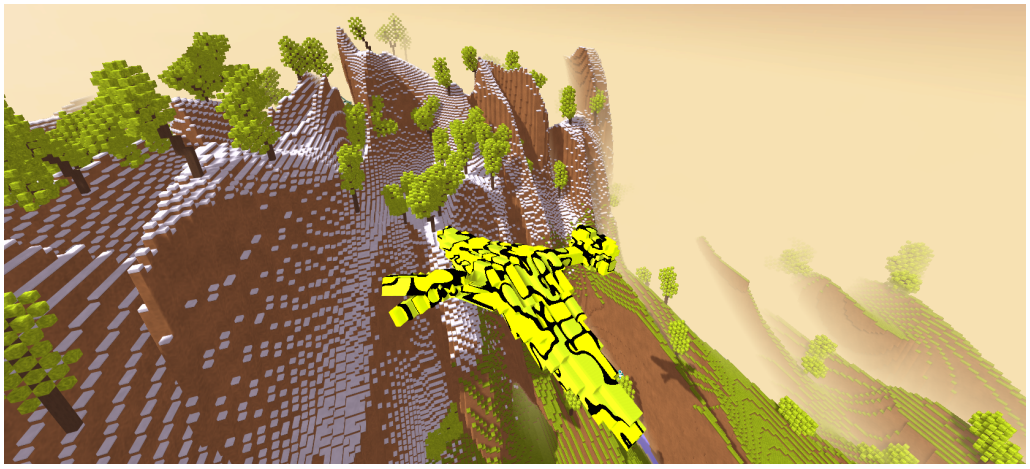


Figure 16: Directional artifacts caused by having corruption influence noise frequency.

The implementation we chose became a middle ground between the other approaches. It requires one extra noise sample per block and it makes the corruption progress gradually. We can also do some biome specific design by giving each biome their own frequency and cutoff point for the corruption noise.

6.2.5 Non-Terrain Generation

Non-terrain generation covers the generation of meshes for trees and animals. Directly representing animals or trees as a `BlockDataMap` is not as easy as it is with terrain, because we need to represent more specific shapes, such as a branch or the head of an animal. We chose to represent trees and animals as a set of line segments instead, because representing a branch as a line makes intuitive sense as opposed to representing it as a 3 dimensional grid, which also makes the code easier to write and maintain. To read about how we generate the line representation of trees read Chapter 6.2.6 and for animals read Chapter 6.2.7.

Listing 6.5: LineSegment struct

```
public class LineSegment {
    public Vector3 a; //Start
    public Vector3 b; //End
    public bool endLine; //Last line in chain?
    public float radius; //Radius of line

    //...
    //methods
    //...
}
```

When we already have a list of `LineSegments`, we have to convert it into a `BlockDataMap` so that the `MeshDataGenerator` can create the mesh data. The core idea for the conversion is to iterate through every block in the `BlockDataMap` and check if the block is within distance of one of the lines. The distance between a block and a `LineSegment` is calculated by using the index of the block as its position, so the block at index (5, 4, 8) is also at position (5, 4, 8), this works because each block is a unit cube. If a block is within distance of a line (distance < line.radius), then the block type will be set to either animal skin, wood or leaf depending on what the line represents. To read about `LineSegment` distance calculations see Chapter 6.2.5.

In order for the above algorithm to work we have to dimension the `BlockDataMap` for the list of line segments, so that it is big enough to contain all of the lines. To do this job we made the `LineSegmentBounds` class, which takes a list of lines and calculates the lower/upper(X, Y, Z) bounds as well as total size of the line list(the total span in coordinate space). We then make a `BlockDataMap` with the total calculated size of the list of lines, and offset every point in the `BlockDataMap` by the lower bounds before doing line distance calculations. The offsetting of lower bounds is necessary because an array can not have negative indexes, but the lines can extend into negative coordinate space.

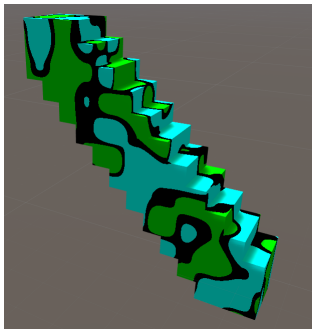


Figure 17: The line (0, 0, 0), (5, 5, 0), radius = 1 converted to a mesh (Line is interpreted as part of an animal)

Tree generation needs some additional functionality, it also has to generate blocks for the leaves, see Figure 18 for an image of trees. This is where the `endLine` attribute of `LineSegment` comes into play, an `endLine` is a line that should generate leaves. The distance check used for leaves generation is: `distance < (line.radius * some multiplier)`, this makes it so that the leaves extend further then the branch. There is an additional check when doing generation of leaves, we also check if every component of the block position vector (converted from float to int) is either odd or even and only generate blocks for strictly even or odd positions. This gives us the alternating block pattern for leaves that can be seen in Figure 18.

We said in the introduction to this chapter that the base building block for our world is a unit cube(see introduction to Chapter 6), which is true for terrain and trees. However, animals use half unit cubes for their meshes, the reasoning for this is to increase the level of detail on animals because we felt like using a full unit cube made them look "blocky". As a consequence of halving the unit cubes, we have to double the size of the `BlockDataMap` in each dimension when we dimension it for the lines representing an animal. We also need to scale the block positions down, so the block from the earlier

example: (5, 4, 8) would be in position (2.5, 2, 4) when interpreting the `BlockDataMap` as a map of half unit cubes. Our code supports generating with any size for the blocks, and the animals themselves can be scaled, so if an animal has a base scale of 1.2, then the final size of the blocks in `BlockDataMap` would be 0.5 * 1.2 = 0.6.

Point LineSegment Distance

The algorithm for calculating the distance between a point and a `LineSegment` is based on an implementation by Dan Sunday [23]. To calculate the distance between a point **P** and a line segment **S** there are 3 cases to consider, **P** is either between **S.a** and **S.b**, or its outside beyond **S.a** or **S.b**. To test if **P** is outside beyond **S.a** we check the dot product of **W = P - S.a** and **V = S.b - S.a**, if the dot product is smaller or equal to 0° then the angle between **W** and **V** is greater then 90, placing **P** outside beyond **S.a**. Once we know that **P** is beyond **S.a** we compute the distance as **dist(P, S.a)**. The process is similar for the case where **P** is beyond **S.b**, we define **W = P - S.b**, **V = S.b - S.a**, if the dot product of **W** and **V** is greater or equal to 0 then the angle between **W** and **V** is less or equal to 90° placing **P** outside beyond **S.b**. The distance for the second case is **dist(P, S.b)**. If none of the past two cases occurred **P** is between **S.a** and **S.b**, the distance can then be calculated by calculating the length of the normal from **P** to the line given by **S**.

6.2.6 Tree Generation

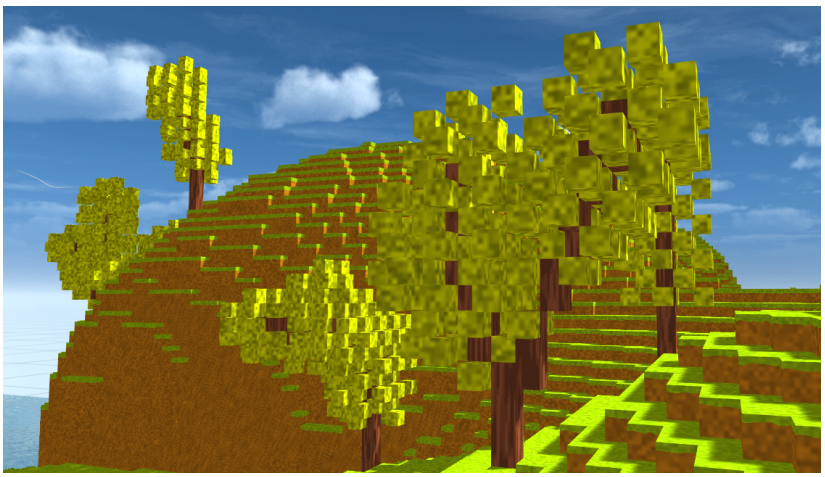


Figure 18: An example of trees

We generate trees using an *L-System* (Lindenmayer system) which is a type of rewriting system [24]. In our implementation we use a *stochastic L-System* in order to generate variety in our trees. L-Systems consists of an *alphabet* and *production rules*. The production rules is a set of rules for how to grow a *word* as you encounter characters from the alphabet. The production rules are applied to an initial string of characters(*axiom*) recursively N times. An example of this would be; Alphabet: { A, B }, Rules: { (A=>AAB) }, this means that the alphabet contains the characters A and B, and when growing the string, all A's that the string grower encounters gets replaced by AAB. So if we start with the string A and treat i recursively 2 times it would grow like this: A => AAB => AABAABB.

Our Implementation

Listing 6.6 shows our stochastic L-System implementation. Since our L-System is stochastic, we have multiple rules for one character, that we choose at random using a random number generator. The character '|' in our production rules delimits the different possible rules that can be chosen for a character. Our stochastic L-System is inspired by the examples from Figure 1.24 in *The Algorithmic Beauty of Plants* [24].

Listing 6.6: Alphabet definition and rules for tree generation

```
private static char[] alphabet = new char[] {
    'N', //Variable
    'M', //Second Variable
    'D', //Draw
    'X', //X axis
    'Y', //Y axis
    'Z', //Z axis
    '+', //Postive rotation
    '-', //Negative rotation
    '[', //Push to stack
    ']'  //Pop from stack
};
private const string start = "DDN"; //Start condition of string (Axiom)
private const float angle = 25f;    //Absolute angle to rotate turtle
private static Dictionary<char, string> rules = new Dictionary<char, string>() {
    // '|' delimits the different rules that can apply to one variable
    { 'N', "[-ZND]+M+XD[-D+XD]N" + "|-YDN-Y" },
    { 'M', "D[+N]-X" }
};
```

We recursively apply the production rules defined in the rules dictionary up to a depth that we pass the recursive string grower function, to create a string representing a tree from the start string(axiom). We control what the trees look like with the production rules that we make, and the recursive depth we use for growing the string. Adding more production rules per character will increase the variety of our trees, and increasing the recursion depth will increase the size, complexity and variety of the trees. There is some trial and error involved in getting good production rules, not all production rules produce nice looking trees. We found some rule of thumb rules for making good production rules, such as keeping the rotations balanced. This means having a somewhat even mix of '+', '-', 'X', 'Y' and 'Z' characters, which are responsible for rotating the lines in the tree. With unbalanced production rules the branches can end up looking like spirals and other unnatural looking shapes, by continuously applying rotations in the same direction. Trees should generally progress vertically, which is why a certain balance is needed to maintain the initial vertical course.

There are two major steps involved in turning the fully grown string into a tree, drawing a tree consisting of line segments using turtle graphics, and turning the list of generated line segments into a voxel mesh.

Drawing the Tree Using Turtle Graphics

We use a struct representing a turtle which maintains a state that is changed by the characters it reads in the string produced by our L-System.

Listing 6.7: Turtle struct for tree drawing

```
private struct Turtle {
    public Vector3 heading; //The direction to draw in
    public Vector3 pos;     //Current position of turtle
```

```
    public Axis axis;        //Axis to rotate turtle in
    public float lineLen;    //Length of the lines the turtle
                             //  draws
}
```

The turtle starts at position (0, 0, 0) with a heading of (0, 1, 0), which means that the turtle is at the tree root looking up. When the turtle encounters a 'D' in the string its processing, it draws a line given by the current state of the turtle, the line drawn starts from the turtles position, and extends `lineLen` units along the turtle heading. The 'X', 'Z', 'Y' characters sets the axis of rotation for the turtle, whereas '+' and '-' applies an actual rotation of 25° on the turtle heading in a negative or positive direction about the current axis. '[' and ']' pushes and pops the current turtle state to a stack, this is what causes branching of our trees. Pushing the turtle to the stack and then having the turtle draw some lines before returning to the previously pushed state by popping gives us multiple subtrees or branches. The last branch that the turtle draws before popping and old state from the stack is set to be a leaf branch, which is a branch that is surrounded by leaves.

Every line drawn by the turtle is added to a list of lines that serves as a representation of a full tree. The list is later fed to a function responsible for turning a set of lines into a voxel mesh, and that mesh is then deployed into the world. To read more about turning lines into a mesh see Chapter 6.2.5.

Making the Tree Generation Deterministic

We want the game world to be persistent, this means that every time the player walks to a certain location, the game world should generate the same way twice. The terrain generation is naturally deterministic as a consequence of being based on noise functions, this is not the case for trees however. We use a random number generator for choosing which rule to apply in our stochastic L-System, and the way we seed this generator is how we make the generation deterministic. We made a function for turning a Vector3 (position) into a seed (any integer), using this we turn the position of the tree into a seed that is used for the generator. The tree position is also calculated in a similar manner, using the position of its parent chunk to seed a generator that is used to generate the position for the tree.

6.2.7 Animal Generation

Figure 19: From left to right: A generated water animal, air animal and land animal

Animals are represented as a set of lines, as mentioned in Chapter 6.2.5, this chapter will cover the generation of the lines representing an animal. The main class responsible for generating the line representation of an animal is the `AnimalSkeleton` class, an abstract super class that the different animal types inherit from. There are tree types of `AnimalSkeletons`; `LandAnimalSkeleton`, `AirAnimalSkeleton` and `WaterAnimalSkeleton`, the animal skeletons are used after generation as well, to read more about animals see Chapter 6.6.

The core idea for generating an animal, is to define a set of attributes for the animal such as number of legs, length of tail, size of head and so on, we refer to these attributes as body parameters. After defining the kinds of body parameters we generate the values for these parameters using a random number generator, this process is known as parametric generation [25]. Once we have the values that define the animal, we interpret those values to draw lines.

Handling Animal Body Parameters

There are 3 main operations that take place regarding the body parameters defining an animal:

1. Define the types of body parameters.
2. Define the ranges for the body parameters.
3. Generate the values for the body parameters.

When we first started implementing animals we had a member variable in `AnimalSkeleton` for each of the body parameters defining an animal, this would result in a solution similar to the below code:

Listing 6.8: Early code for animal defining attributes

```
public class AnimalSkeleton {
        float legLength;
        float legJointLength;
        int legJoints;
        int legPairCount;

        //...

        private void generateBodyParams(){
                legLength = rng.randomFloat(minLegLength, maxLegLength);
                legJoints = rng.randomInt(minLegJoints, maxLegJoints);
                legPairCount = rng.randomInt(minLegPairCount, maxLegPairCount);
                legJointLength = legLength / legJoints;
                //...
        }

        //...
}
```

There are some problems with the above solution, the first being that it requires a large amount of member variables to define an animal. The Air animal for instance is defined by 19 body parameters, which would yield 19*3 = 57 individual member variables for the `AirAnimalSkeleton` class. Handling the body parameters in this manner becomes hard to maintain, when you add a new body parameter you have to remember to define the ranges for it, and put code for generating the value in the `generateBodyParams` function. Finding a body parameter you want is also hard because you would have to remember its name, the above implementation gives no logical grouping of the body

parameters.

We refactored the above implementation before we started to implement more then one type of animal. The new implementation logically groups the body parameters into a dictionary and groups the allowed ranges for each body parameter in a second dictionary. Each body parameter has its own enum, which is the key for the dictionary. Since the body parameters is a mix of floats and integers the dictionaries are a custom dictionary type we wrote called `MixedDictionary`, which uses template getters allowing us to specify the type for the parameter we are getting. The ranges are also represented using a `Range<T>` class, with a minimum and maximum value for the range. The above implementation after the refactor looks like this:

Listing 6.9: Current code for animal defining attributes

```
public enum BodyParameter {
    //...
    SPINE_LENGTH, SPINE_JOINTS, SPINE_JOINT_LENGTH, SPINE_RADIUS,
    //...
}
public abstract class AnimalSkeleton { //Abstract super class
    //...
    protected MixedDictionary<BodyParameter> bodyParametersRange;
    protected MixedDictionary<BodyParameter> bodyParameters;
    //...
}
public class WaterAnimalSkeleton : AnimalSkeleton { //Body parameters are defined in derived classes
    public WaterAnimalSkeleton(Transform root, int seed = -1) {
        bodyParametersRange
            = new MixedDictionary<BodyParameter>(new Dictionary<BodyParameter, object>() {
                //...
                { BodyParameter.SPINE_LENGTH, new Range<float>(7, 17) },
                { BodyParameter.SPINE_JOINTS, new Range<int>(3, 7) },
                { BodyParameter.SPINE_RADIUS, new Range<float>(1f, 2.0f) },
                //...
            }
        );
       //...
    }
    //...
     override protected void generateBodyParams() {
        base.generateBodyParams(); //Iterates through bodyParametersRange and generates bodyParams
        bodyParameters.Add(
            BodyParameter.SPINE_JOINT_LENGTH,
            bodyParameters.Get<float>(BodyParameter.SPINE_LENGTH)
            / bodyParameters.Get<int>(BodyParameter.SPINE_JOINTS)
        );
    }
    //...
}
```

With the new system we can define a new animal such as the `WaterAnimalSkeleton` by populating the `bodyParametersRange` dictionary, the `generateBodyParams` function will then iterate through the `bodyParametersRange` dictionary and generate all of the body parameters automatically. The automatic generation of the body parameters gives us less code to maintain, however we still have to write code for the body parameters that are calculated instead of generated. The main drawback of the refactor is that it makes accessing a body parameter more verbose, but this is outweighed by having the attributes logically grouped and automatically generated.

Drawing Lines from Body Parameters

After the body parameters have been defined and generated they get used for generating the lines that represent the `AnimalSkeleton`. The `AnimalSkelton` superclass defines an abstract method `makeSkeletonLines()` which the derived classes override. We found no clever method to automatically generate a skeleton from a set of body parameters so we pretty much just spell it out in code. We start by drawing the line segment for the spine of the animal, the center point of the spine is at (0, 0, 0), then we draw the other lines from the spine so that the entire skeleton is connected. We do use loops for generating the legs for instance, since the only difference between the up to 3 right legs of a `LandAnimal` is where they start in the spine line.

The lines are organized into a dictionary in a similar fashion to the body parameters. The key for the dictionary of skeleton lines is the `BodyPart` enum (see Listing 6.10 for definition), this allows us to address specific lines in the skeleton which comes in handy when doing mesh skinning and animation.

Listing 6.10: Skelton lines dictionary

```
public enum BodyPart {
    ALL = 0,
    HEAD,
    NECK,
    SPINE,
    RIGHT_LEGS, LEFT_LEGS,
    TAIL,
    RIGHT_WING, LEFT_WING
}
public abstract class AnimalSkeleton {
    //...
    protected Dictionary<BodyPart, List<LineSegment>> skeletonLines;
    //...
}
```

Listing 6.11: Creating neck line for land animal

```
override protected void makeSkeletonLines() {
    //...
    //NECK
    LineSegment neckLine = new LineSegment(
        spineLine.a,                                          //Start of line
        spineLine.a + new Vector3(0, 0.5f, 0.5f).normalized *  //End of line
        bodyParameters.Get<float>(BodyParameter.NECK_LENGTH),
        bodyParameters.Get<float>(BodyParameter.NECK_RADIUS)    //Radius of line
    );
    addSkeletonLine(neckLine, BodyPart.NECK);
    //...
}
```

In Listing 6.11 you can see how the land animal neck line is generated, it starts at the begging of the spine, and extends in the (0, 0.5, 0.5) direction for the length of the neck line. The `addSkeletonLine` function adds the line to the `skeletonLines` dictionary, the line gets added to the `BodyPart.ALL` key and the specified `BodyPart.NECK` key.

The next step after all of the lines have been defined is turning it into a mesh, which is explained in Chapter 6.2.5.

Bones and Mesh Skinning

The generation of animals is not done after the mesh has been created as with terrain and trees. The last step in animal generation is creating the animation bones of the mesh

and binding them to the mesh. The motivation behind doing this is that unlike trees and terrain we want the animals to move. This last steps enables us to bring the animal meshes to life, if we create a bone for one of the legs of the animal, and we bind it correctly to the mesh, then we will be able to rotate the bone and the mesh will deform appropriately, see Figure 20 for an image of bone rotations.

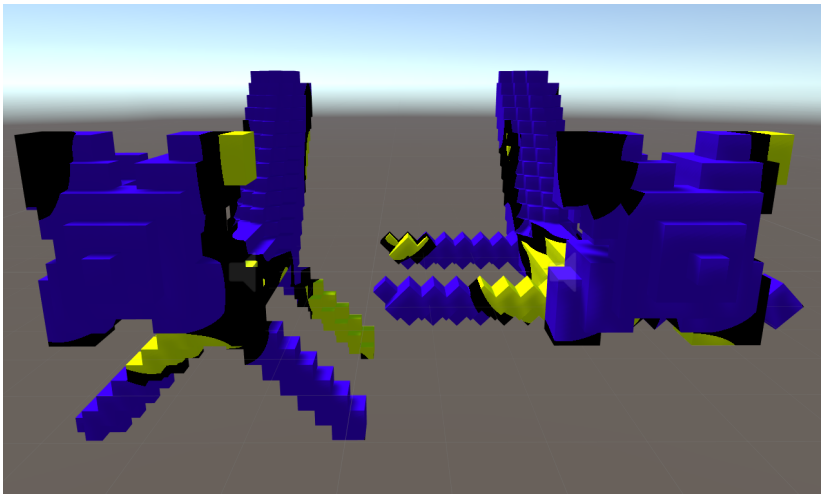


Figure 20: the animal to the right has no rotations applied to animation bones, the animal to the left has a 45 degree rotation about the z axis applied to all of its legs.

In Unity an animation bone is a `GameObject` with only one component, the transform component [26]. The animation bones are put into a hierarchy of bones making up an animation skeleton, when you apply a rotation to a certain bone that rotation is in turn applied to all of the children of that bone. The property of propagating a rotation downwards in the hierarchy is important for creating logical animations. Consider the case where you rotate the neck, you would expect the head to also follow the rotation of the neck since its connected to the neck.

There are two main operations in making the animal mesh animation ready:

1. Creating a hierarchy of bones making up an animation skeleton.
2. Binding the animation skeleton to the mesh.

The animation bones are created and stored in a similar fashion to the skeleton lines described in the above sub section, in a dictionary with `BodyPart` as the key. To create all of the animation bones the `makeAnimBones()` function is called, similar to the `makeSkeletonLines()` function for making skeleton lines. In Listing 6.11 we showed how the neck line for the skeleton was created, Listing 6.12 shows how the animation bones for the neck is created.

Listing 6.12: Creating neck bones for land animal

```
override protected void makeAnimBones() {
    //...
    //NECK
        Bone neckBoneBase = createAndBindBone(
            skeletonLines[BodyPart.NECK][0].a,  //Position of bone
            spineBone.bone,                     //Parent bone of bone
            skeletonLines[BodyPart.NECK][0],    //Line used for skinning the bone
```

```
            "Neck",                             //Name of bone
            BodyPart.NECK                       //Body part of bone
        );
        neckBoneBase.minAngles = new Vector3(-90, -90, -90);
        neckBoneBase.maxAngles = new Vector3(90, 90, 90);
        createAndBindBone(
            skeletonLines[BodyPart.NECK][0].b,
            neckBoneBase.bone,
            skeletonLines[BodyPart.HEAD],
            "Neck",
            BodyPart.NECK
        );
    //...
}
```

Even though the bones themselves are singular points, we need to provide the skeleton line that the bone is associated with when creating and binding the bone because we want the vertices to be tied to the line of the bone and not the position of the bone. The process of binding vertices in the animal mesh with bones is done by iterating through every vertex in the mesh and calculating which skinning line the vertex is closest to. When we find the closest skinning line we assign the vertex to the bone that owns that skinning line. Once a vertex has been assigned to a bone all rotations applied to the bone will also be applied to the vertex, giving us the ability to animate and move the mesh of the animal.

6.3 WorldGenManager

The `WorldGenManager` is the class responsible for handling the chunks and the animals in the world. It decides where in the world a chunk or animal should be generated or removed. The `WorldGenManager` considers the player as the center of the world and generates chunks around the player. Chunks and animals are pooled, so when something is removed it is placed back into a pool instead of being destroyed. It does not handle the actual generation of chunks and animals, that is handled by the `ChunkVoxelDataThread` threads. When the `WorldGenManager` decides that a chunk should be generated at a position it sends an order to the `ChunkVoxelDataThread` threads, when the threads are done generating they send the result back to the `WorldGenManager` which then deploys the generated content into the world. To read more about the `ChunkVoxelDataThread` threads see Chapter 6.4. To read more about the technical design of the `WorldGenManager` see Chapter 4.2.

The `WorldGenManager` handles these task in an update loop seen in Listing 6.13.

Listing 6.13: WorldGenManager update

```
void Update() {
    offsetWorld();
    clearChunkGrid();
    updateChunkGrid();
    orderNewChunks();
    consumeThreadResults();
    handleAnimals();
}
```

6.3.1 Handling Chunks

There are 3 main variables controlling the logic of placing chunks into the world: `chunkSize`, `chunkCount` and `playerPosition`. `chunkSize` is the length of one side of a chunk,

`chunkCount` is the amount of chunks in a given dimension (x or z), so the total amount of chunks is $chunkCount^2$. `playerPosition` is the position of the player. This means that the game manages a $chunkCount^2 * chunkSize^2$ area of the world at any given point. The player is at the center of the managed area. The valid positions of chunks is given as $(chunkSize * n, chunkSize * m)$, where n and m can be any integer value. This makes it so that chunks align with each other perfectly without creating gaps or overlaps.

In order to manage the $chunkCount^2 * chunkSize^2$ area centered on the player we use a multidimensional array of chunks `chunkGrid`. To manage all active chunks in general we use a `activeChunks` list. We need a separate list of chunks because chunks change their position in the `chunkGrid` as the player moves and at some point chunks also fall out of the bounds of the `chunkGrid`.

The `chunkGrid` is maintained by mapping every chunk in the `activeChunks` list to indexes in the `chunkGrid`. To do the job of mapping between positions of chunks and their index in the `chunkGrid` we made two helper functions: `Vector3Int world2ChunkIndex(Vector3 worldPos)` and `Vector3 chunkIndex2world(Vector3 chunkPos)`. The two functions are the inverse of each other, only the x and z components are used for indexing. The chunk index is calculated from the chunk world position as follows:

```
Start by chunkSize normalizing the player position,
so that the player position becomes a
valid chunk position in world space.
playerPos = floor(playerPos/chunkSize) * chunkSize
Subtract the playerPos from the chunk world position.
chunkIndex = chunkWorldPos - playerPos
If chunkWorldPos == playerPos in this case then the index
would be (0, 0, 0), we need to subtract an offset
to center the chunkGrid on the player.
offset = (chunkCount/2 * chunkSize, 0, chunkCount/2 * chunkSize)
chunkIndex = chunkIndex - offset
The last step is dividing by chunkSize and converting to int.
chunkIndex = floorToInt(chunkIndex/chunkSize)
```

Using the above methods we can maintain the `chunkGrid`. For every update we clear the `chunkGrid` first, meaning that we set every element to `null`. Clearing is needed because player movement will invalidate the `chunkGrid`, as it is meant to map chunks relative to the player position. After clearing we update the chunk grid by iterating through the `activeChunks` list mapping the chunks to an index in `chunkGrid`. If a chunk produces an index that is out of bounds for `chunkGrid` it is removed from `activeChunks` and put back into a pool.

After mapping existing chunks to the `chunkGrid` we iterate through the `chunkGrid` looking for `null` elements. An element being `null` is what triggers the `WorldGenManager` to order new chunks from the worker threads. The data required by the threads for a chunk order is the world position of the chunk (used to determine seed for noise sampling, see Chapter 6.1.1). We use `Vector3 chunkIndex2world(Vector3 chunkPos)` mentioned earlier for mapping the index of the `null` chunk to a world position, the calculated position is then used in the order sent to the threads. We also keep track of what chunks we have ordered in a `pendingChunks` hash set to prevent us from ordering the same chunk multiple times while the threads work.

Orders completed by the worker threads appear in a thread safe queue. When chunks are completed and dequeued from the thread results queue one of two things happen. The chunk either gets deployed into the world immediately or it gets placed into a waiting list. Chunks that are far away from the player end up in the waiting list, this is because the `chunkGrid` covers an area larger then the player can see. There is no point in using CPU time deploying and maintaining a chunk that wont be seen by the player. The reason for even generating these chunks to begin with is that world generation is time consuming and the waiting list acts as a buffer reducing the chance that the player could run out of chunks. Once in a waiting list one of two things happen to the chunk, it either gets within range of the player at which point it is deployed, or it gets outside the range of `chunkGrid` in which case it is discarded. Whenever a chunk is deployed or discarded it is removed from the `pendingChunks` hash set so that it can be generated again if the player returns to the same area.

6.3.2 Handling Animals

Our first approach to handling logic for where and when to generate animals was based on maintaining a set of 20 animals. We would generate a new animal whenever one of the animals ended up outside the range of the managed world. The generation was done by reusing the out of range animal for a newly generated animal. The position of the newly generated animal was calculated by generating in range positions that were also inside the bounds of an existing chunk. This approach produced unfavourable distributions of animals, the managed area of the world is not usually completely filled with chunks. This makes it so that the animals gets focused on the area of the world that is generated when they are generated.

To address the animal distribution issue we made animals generate on a per chunk basis instead. Whenever a chunk is deployed into the world there is an 8% chance that an animal will be generated for that chunk. The value 8% was found through testing what worked well. Tying animal spawning to chunk spawning makes it so that the number of animals in the world scales with the number of generated chunks, whereas it was constant in the previous approach. This makes it so that the distribution of animals remains the same regardless of how many active chunks there are.

When a chunk triggers the generation of an animal the center of the chunk is used as the animal position. There are 3 animals types, water, land and air animals (see Chapter 6.6), so we have to decide which one to generate. This is done by using the `VoxelPhysics` class (see Chapter 6.5), we ray cast down from the animal spawn position. If we find water by ray casting we spawn a water animal, if not we spawn a land or air animal both having equal chance of being spawned.

Animals are generated by the same threads that generate chunks of the world. The thread orders for animals need the animal position and an animal skeleton. The threads will then generate the provided animal skeleton and return it in the same thread safe queue as used for chunks. When the manager receives the animal skeleton from the threads it is applied to an animal which is then deployed into the world. To read more about animal skeletons see Chapter 6.2.7.

Animals are removed from the world by being placed back into a pool for future use when they end up outside the range of the managed world.

6.3.3 On-The-Fly Shifting of Coordinates

The `WorldGenManager` also does what is known as On-the-fly shifting of coordinates [27], which is the first function call in Listing 6.13. The shifting of coordinates is done in order to prevent issues caused by limited floating point precision in large or infinite game worlds. These issues could be physics bugs or rendering bugs. On-the-fly shifting of coordinates works by translating the entire world back to the world origin whenever some distance is beyond some threshold. We use the distance between the player and the origin to work out when to do the coordinate shift. World generation is controlled by sampling noise from coordinates in world space, so we when ordering chunks we need to provide the positions the chunks would have had if no coordinate shifting took place. To achieve this we store the total world shift applied to the world in a variable which we factor in when setting chunk positions in thread orders. This effectively hides the fact that coordinate shifting is happening from the chunk and animal generation systems. See Figure 21 for an image of the effects of not accounting for the coordinate shift when generating the world.

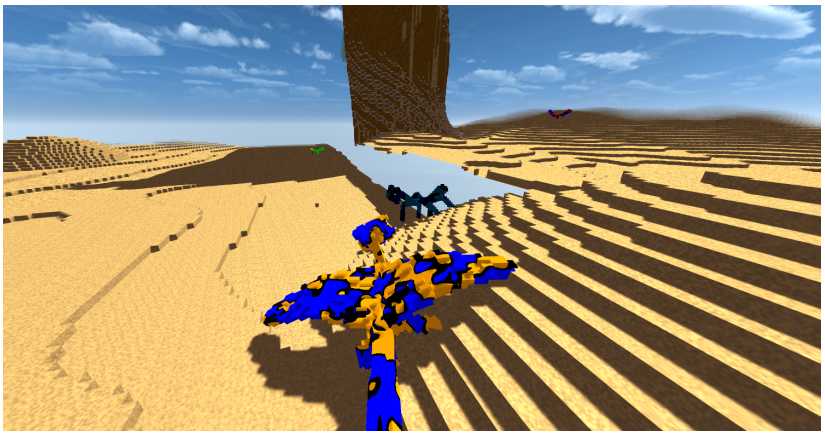


Figure 21: The image shows what happens to the world when not accounting for the coordinate shifting when generating the world. The gap in the world is produced by neighbour chunks generated before and after the coordinate shift. This makes it so that they sample noise from vastly different coordinates, when they should be sampling from the same area.

6.4 Multithreading

Due to the high computational cost of generating the terrain in real time, we had to multithread the world generation system. Procedural content generation of terrain based on noise functions is a workload that can be threaded with little consideration. This is because every chunk(see Figure 5) of the world is logically independent of each other. The result of generating one chunk has no impact on any other chunk. This means that we can make N threads all of which generate their own separate piece of the world at the same time without conflict. However, making the threaded generation scale well across large thread counts is not as easy as just making a lot of threads, it requires good memory management. See Chapter 7 for more information about multithreaded performance.

6.4.1 Our Multithreading Implementation

Our multithreading implementation revolves around the `WorldGenManager`(see Chapter 6.3) class, which is the class responsible for handling our world generation. The `WorldGenManager` creates a set of N threads(based on system thread count) that it uses for world generation at the start of the game. The threads themselves are instances of a class `ChunkVoxelDataThread`, with the functionality needed for carrying out orders and generating content for the world. The generation work carried out by the threads can be read about in chapter 6.2.

Thread Communication

There are 2 key classes and 2 key data-structures involved in the thread communication. The classes are `Order` and `Result`, they contain the data needed for handling chunk or animal generation and an enum denoting which one to generate. The two data-structures are seen in Listing 6.14, every thread as well as the `WorldGenManager` contains references to these two data-structures. The `WorldGenManager` adds orders to the `BlockingList` of orders, then the threads pick an order from the list to execute. The reason for using a list instead of a queue is that different orders have different priorities as we will explain later. Once a thread has completed an order they make a result object that is then enqueued to the `LockingQueue`. The reason for using a non blocking queue for the results is that we do not want the `WorldGenManager` to block when getting results. The `WorldGenManager` runs on the main thread that our game runs on, blocking the manager would cause the entire game to freeze. We instead check if the count of objects in the queue is greater then 0 before dequeuing. Doing this is safe from race conditions because the `WorldGenManager` is the only object dequeuing from the queue. If the number does change between the check and dequeuing then it can only go up from a worker thread enqueuing a new result.

Listing 6.14: Thread communication data-structures

```
private BlockingList<Order> orders;
private LockingQueue<Result> results;
```

Thread Order Processing

Order processing by threads starts by selecting which order to carry out. Every order comes with the position of the chunk or animal that is going to be generated. The threads prioritize generation of orders that are likely to be seen by the player. To achieve the prioritization the threads iterate through every order and calculate a score for the order. An order is scored based on its distance from the player and the angle between the (order position - player position) vector and player movement direction + looking direction vector. A low distance and a low angle gives a favourable score, the thread executes the order with the best score. There is a special case where an order will be selected regardless of score, which is when an order is out of bounds, meaning that its position is outside the current relevant area of the world. These orders are cancelled and never completed.

The `WorldGenManager` usually produces orders at a rate that is faster then what the `ChunkVoxelDataThreads` can keep up with. Prioritizing orders in this way makes it so that the player is less likely to outrun the world generation. Before we implemented the order prioritization system the threads would execute orders in a FIFO order. This lead

to issues with the world generation not keeping up with the movement of the player. It also caused issues when the game started because the player would not have any terrain to stand on in the beginning.

Once an order has been selected the thread carries out the generation as described in Chapter 6.2 and sends the result back to the `WorldGenManager`.

Unity Challenges with Threading

The Unity API is not thread safe [28], and Unity does not officially support the use of multiple threads, which leads to us having to do some of the workload we would like to do in the thread in the main thread instead. An example of such a workload is: Creating a new instance of the `Mesh` class (Built-in Unity class for meshes) in a thread. The `Mesh` class is part of the Unity API and as such we are not allowed to instantiate it in threads. We can however do everything but instantiating in the thread, all the data that the `Mesh` class needs such as vertices, uvs, normals can be created in the thread, so we generate all of the data needed for a `Mesh` and send that back to the main thread, which uses it to create the final `Mesh`.

Unity prevents the inclusion of `C#` concurrent data-structures, so we had to make our own thread safe data-structures which can be seen in Listing 6.14.

6.4.2 Other Multithreading Implementations We Considered

We did at some point in the development process feel like the above threading implementation made our code less elegant than it could be, because we had to go through the `WorldGenManager` class whenever we wanted to get some threaded workload done. We considered using a thread pool instead, in an effort to decentralize our threading implementation. With the thread pool we could simply send `C#` function delegates to the thread pool if we wanted to use it, instead of having the `WorldGenManager` send a work order to its threads, which felt like "jumping through hoops" at the time.

However, when we did implement thread pooling, we actually found it much harder to write readable and maintainable code, with the decentralized threading model it became much harder for the `WorldGenManager` to keep track of the world generation state. When chunks of the world were responsible for generating themselves instead of having to go through the `WorldGenManager`, the manager had no insight into the generation progress. With the above implementation the `WorldGenManager` would always know what aspects of the world that were done generating and not, because they would appear in the `LockingQueue` as they finished generating. This system is really convenient for us because we want the `WorldGenManager` to have all the authority in a world generation context, because it is the `WorldGenManager`'s job to delegate and deploy all aspects of the world.

6.5 Voxel Physics

The voxel physics system allows the animals to interact with the terrain. It consists of two classes: `VoxelPhysics` and `VoxelCollider`. There are two key features provided by the system, *collision detection* and *ray casting*. It was introduced as an optimization which you can read more about in Chapter 7.5.

6.5.1 Voxel Physics Class

The `VoxelPhysics` class is a static class providing functionality for querying the terrain. It is inspired by Unity's own `Physics` class. The main functionality we want from `VoxelPhysics` is ray casting as provided by Unity's `Physics.Raycast(...)` [29]. The most important function provided by `VoxelPhysics` is `voxelAtPos(Vector3 worldPos)`, it takes a world coordinate as input and returns the block type at that position. This effectively allows us to index space, which is the functionality that the `VoxelCollider` and `VoxelPhysics.Raycast(...)` is based on.

Voxel at Position

The `VoxelPhysics` class maintains a reference to the `WorldGenManager` (see Chapter 6.3) to make `VoxelPhysics.voxelAtPos(Vector3 worldPos)` work. The chunks managed by the `WorldGenManager` keep a reference to the `BlockDataMap`(see Chapter 6.2) that they are based on. `voxelAtPos(Vector3 worldPos)` works by indexing the `BlockDataMap` of the chunk that the world position falls inside. The `WorldGenManager` has a function that can map a world position to a chunk, using that function with the provided position from calling `voxelAtPos(Vector3 worldPos)` gives us the chunk to index. To find the index of the block inside the given chunk we subtract the position of the chunk from the provided position. We then index the chunk's `BlockDataMap` and return the block type for the indexed block. In the event that the function fails to find any chunks at the provided position `BlockType.NONE` is returned. This can be caused by an out of bounds position, or the lack of a chunk at a position.

Ray casting

The Unity ray cast is a method of finding colliders in world space. You define a ray with an origin and a direction, then the ray cast function returns information about the cast and the `RaycastHit` if the ray cast hit anything. The `RaycastHit` contains information about what was hit, the position of the hit and more.

Our `VoxelPhysics.Raycast(...)` function casts rays against the voxel terrain using the `VoxelPhysics.voxelAtPos(Vector3 worldPos)` function. There are 3 types of targets to cast against as seen in Listing 6.15.

Listing 6.15: Voxel ray cast targets

```
public enum VoxelRayCastTarget {
    SOLID,      //Block such as dirt and sand
    NON_SOLID,  //Water, wind, none
    WATER       //Only water
}
```

When the ray cast hits something it returns a hit object `VoxelRayCastHit` as seen in Listing 6.16.

Listing 6.16: Voxel ray cast hit

```
public class VoxelRayCastHit {
    public BlockData.BlockType type;//block type of hit block
    public Vector3 blockPos;        //Position of hit block
    public Vector3 point;           //Position of hit
    public float distance;          //Distance of hit to ray origin
}
```

The `VoxelPhysics.voxelAtPos(Vector3 worldPos)` function makes the implementation of `VoxelPhysics.Raycast(...)` simple. We sample voxels along the provided ray

using a for loop going from 0 to the max length of the ray. We then calculate the current position along the ray as `Vector3 sample = ray.origin + ray.direction * t;` where `t` is the for loop variable. `t` is incremented by 1 because we use unit cubes for the terrain. In the event that the ray is diagonal to the coordinate system we should increment by $\sqrt{2}$ instead since that is the diagonal length of a unit cube. However, sampling a block more then once has no bearing on the result of the ray cast, so we use a delta of 1 for `t` regardless of direction for the sake of simplicity. If the ray cast finds a block matching the target it uses its current `sample` position and the current block to make a `VoxelRayCastHit`. If no matching blocks are found it returns a `VoxelRayCastHit` with the type set to `BlockType.NONE` to indicate not hitting anything.

6.5.2 Voxel Collider Class

The `VoxelCollider` class is made for use with animals. It causes the animals to collide with the terrain and calls 3 event based functions in the `Animal` class. The 3 event functions are seen in Listing 6.17. They are similar to 3 collision event messages provided by Unity, which are `OnCollisionEnter`, `OnCollisionStay` and `OnCollisionExit`. The animals need this information for their own logic, to read more about animals see Chapter 6.6.

Listing 6.17: Voxel collider events

```
animal.OnVoxelStay(lastVoxel);
animal.OnVoxelEnter(voxelAtPos);
animal.OnVoxelExit(lastVoxel);
```

To handle collisions the `VoxelCollider` relies on `VoxelPhysics.voxelAtPos(Vector3 worldPos)`. An animal is considered to have collided with the terrain if the voxel at the animal position is a solid block. The collisions are handled in a preventative manner, the collider checks if the animal will collide in the next update with its current velocity. To prevent a collision we need to change the velocity such that the collision in the next update does not occur. We could set the velocity to zero, but this would immobilize the animal giving the animals a very poor usability for the player. Instead we do the collision prevention on a per axis basis for the velocity vector. We build the new velocity one axis at a time with data from the current velocity vector. Each time we add a value to an axis of the new velocity vector we run the collision check to see if the new velocity would cause a collision. If the axis added to the new velocity vector will cause a collision it is removed. This way we only stop the velocity for the components of the velocity vector causing a collision instead of completely immobilizing the animal.

6.6 Animals

The game features 3 different types of animals: `LandAnimal`, `AirAnimal` and `WaterAnimal`, these can be seen in Figure 19. The design of an animal is separated into 3 separate classes as described in the technical design (see Chapter 4.3). The classes are `Animal`, `AnimalBrain` and `AnimalSkeleton`. The `Animal` class implements the *functionality* of an animal. The functionality is the animals ability to function in the environment, such as the ability to walk in the terrain. The *behaviour* is implemented by the `AnimalBrain`, behaviour controls how the functionality i used, by for instance deciding which direction to walk. The `AnimalSkeleton` implements the *body* of the animal, and allows the animals to access the body for other functionality, such as animations. `AnimalSkeleton` is described

in detail in Chapter 6.2.7.

6.6.1 Animal State

Animals have a state that influence their behaviour and functionality. This makes the data describing the state of an animal relevant for both the animal functionality (`Animal`) and the animal behaviour (`AnimalBrain`). For this reason we made a class `AnimalState` to contain the state of the animal. Both the `Animal` and `AnimalBrain` class keep a reference to the `AnimalState` of the animal. See Listing 6.18 for the implementation of the animal state class. The `desiredSpeed` and `desiredHeading` are set by the `AnimalBrain`, then the `Animal` class responsible for functionality, tries to execute the behaviour. The various boolean flags describe various states of the animal. The states are not exclusive, it is for instance possible to be in water while being able to stand.

A criticism of the `AnimalState` implementation is the lack of protection for the member variables. The `AnimalBrain` class is the only class that modifies the `desiredSpeed` and `desiredHeading` members, and the `Animal` class is the only class that modifies all of the other members. With the current implementation any class can modify any member of `AnimalState`, which is something we wanted to refactor but did not get to.

Listing 6.18: Terrain shader arrays

```
public class AnimalState {
    public bool onWaterSurface = false;
    public bool inWater = false;
    public bool grounded = false;
    public bool inWindArea = false;
    public bool canStand = false;

    public float desiredSpeed = 0;
    public float speed = 0;

    public Vector3 desiredHeading = Vector3.zero;
    public Vector3 heading = Vector3.zero;
    public Vector3 spineHeading = Vector3.forward;

    public Transform transform;
}
```

Determining State

The voxel physics system described in Chapter 6.5 is used to determine the state of an animal. The `OnVoxelStay(BlockType type)` function called by the animals `VoxelCollider` is used to determine the values of `inWater` and `inWindArea`. Since water and wind blocks do not cause collisions the animals can stay inside these blocks. If the animal is inside a water block `inWater` is set to true and the same goes for wind blocks and `inWindArea`. `onWaterSurface` is true if the animal is not in water while the block directly below the animal is water.

We use the ray casting provided by `VoxelPhysics` to determine `canStand` and `grounded`. We cast a ray starting at the animal position in the local down direction for the animal. If the ray hits nothing then the animal is not grounded and cant stand, if the ray hit a solid block the animal may be grounded. We check the distance from the ray origin to the point where the ray hit something to work out if the animal can stand and or is grounded. Land and Air Animals have a `stanceHeight` based on the length of their legs, which is the distance between the animal spine and ground when standing. If the distance to the

ground is less then or equal to the `stanceHeight` then the animal can stand. The animal is considered grounded if its distance from the ideal `stanceHeight` is less then or equal to the `stanceHeight`, the reasoning for this is explained in the next section.

6.6.2 Making Animals Functional

The animal functionality is implemented in the update method of the derived classes: `LandAnimal`, `WaterAnimal` and `AirAnimal`, using methods from the base class or custom methods for the specific animal. There are 4 activities which are done by every animal type and one last that is done only by land and air animals:

1. Calculating speed and heading.
2. Calculating velocity.
3. Gravity calculations.
4. Handling of animations.
5. Leveling the animal with the terrain.

Calculating Speed and Heading

`speed` and `heading` is calculated based on the `desiredSpeed` and `desiredHeading`. The `speed` will accelerate towards the `desiredSpeed` at a certain rate depending on the current state of the animal. Acceleration is slower when in water than when on land for instance. The `heading` of the animal will rotate at a certain rate towards the `desiredHeading`. When the speed or heading are within set thresholds of their goal values they stop changing.

Calculating speed and heading in a gradual manner gives the animals more natural behaviour. If the player decides to turn around, the animal should not instantly reverse its heading but gradually turn into its new heading instead in a fluid motion.

Calculating Velocity

The velocity calculation uses the `speed` and `heading` or `spineHeading` with the gravity acting on the animal to calculate the final velocity. The calculation is as follows: `finalVel = speed * heading + gravity`. The difference between `heading` and `spineHeading` is that `heading` is a horizontal direction whereas `spineHeading` includes the vertical axis as well. The `spineHeading` is used for following the terrain when walking on the ground, otherwise `heading` is used.

Gravity Calculations

The gravity of animals are not true to real world physics. It is used more like a tool to control their vertical position. Depending on the state of the animal one of 4 gravity calculations is used:

1. Grounded gravity: Calculates gravity in such a way that the animals height above the ground matches the animals `stanceHeight` mentioned in Chapter 6.6.1. It uses less force then normal gravity because it needs finer control to move the animal into position. It is used when the animal can stand or when the animal is grounded. It will apply an upwards force on the animal when the animal is below the `stanceHeight` and a downwards force when the animal is above. This is why the animal can be considered grounded even if not being able to stand, because we need the grounded gravity to bring the animal into the `stanceHeight` even if the

animal is above the stance height.
2. Not grounded gravity: This works like normal gravity and is used when the animal is not in water, not on the water surface, not grounded and can not stand. For air animals the force of gravity approaches zero as they approach their max speed, this helps them stay airborne.
3. Water gravity: Applies an upwards force to the animal to push them to the water surface, it is used when in water while not being able to stand. If the animal can stand the grounded gravity calculation is used, this enables the animal to walk out of the water and onto land. For Water animals it works differently, zero force is applied in their case. This allows the water animals to move around inside the water.
4. Water surface gravity: Applies zero gravity to the animal, so that they float on the water surface. It is used when the animal is on the water surface while not being able to stand for the same reasoning as with water gravity. It is not used for water animals, as they can swim inside the water.

The reason for using gravity to control the animal height above the terrain is to make the physics and animations easier to implement. If the animals height above the ground were to be the result of collisions between the animal feet and the terrain then that would make the animations impact functionality. Walking animations move the feet, and we do not want this to cause actual motion of the animal as a whole. We want to separate the representation of the animal from the functionality of an animal. This means that our animations are purely visual with no function beyond that. The animations makes it look like the animals are walking on the ground, but they are actually just hovering above the ground as far as our physics is concerned. To read more about animations for animals see Chapter 6.6.4.

Handling of Animations

The animation handling decides which animation to play for the animal depending on state. For Air animals for instance this means playing a flying animation when not grounded, or a walking animation when grounded. The animation system supports transitioning from one animation to another which is used when switching animations.

Leveling the Animal with the Terrain

Leveling the animal with the terrain means that we set the rotation of the animal spine in such a way that the animal aligns with the inclination of the terrain it is standing on. This process is responsible for calculating the `spineHeading` from Listing 6.18 and rotating the animal so that they also look like they are following the terrain. This enables animals to move up or down mountains or any other inclination. See Figure 22 for the effects of spine leveling.

We rotate the spine in two steps by matching it with the terrain inclination in two axes to do the leveling. The first axis is the horizontal `heading` of the animal, the second is the horizontal normal vector of the `heading`, these axes are the animals local x and z axis. This makes it so that the animal spine will both pitch and roll to match the terrain. To match the spine in one axis we need a direction for the terrain to compare against. To find the direction of the terrain we use the provided axis and ray casting. We cast one ray to find the ground at both sides of the animal along the line given by the animal

Figure 22: The effects of leveling an animal with the terrain. The animal spine is parallel with the steep incline.

position and the provided axis. The distance between these points is given by the length of the spine when pitching and the length of the legs when rolling. When we have our two origin points for ray casting we cast a ray down or up to find the surface level of the ground. This gives us two surface points on the ground that we can use to make a vector representing the direction of the terrain under the animal along the provided axis.

When we have the direction of the terrain we have to rotate the spine to match the terrain vector. To find the angle of the rotation we calculate the angle between the terrain direction and the direction of the spine in the provided axis. To find what axis to rotate the spine around we calculate the cross product of the two directions to get their normal vector. The rotation is then applied to the animal spine about the normal vector. We multiply the angle of rotation by some constant to prevent the animal from doing sudden unnatural movements when leveling with the terrain.

6.6.3 Giving Animals Behaviour

The `AnimalBrain` is the class responsible for the behaviour of animals. As can be seen in Figure 3 there are two main categories of `AnimalBrain`, `AnimalBrainPlayer` and `AnimalBrainNPC`. `AnimalBrainPlayer` implements it by listening to player input whereas `AnimalBrainNPC` implements behaviour through AI.

The `AnimalBrain` is responsible for 3 things: Setting the `desiredHeading` and the `desiredSpeed`, and invoking special actions. The special actions for water and land animals is jumping and for air animals it is taking off for flight. The `AnimalBrain` has a dictionary of function delegates `actions` which contains all actions that the animal can do. The implementation of these actions are handled by the `Animal` class or its derived classes. The `AnimalBrain` is only responsible for deciding when to invoke them.

NPC Brains

The AI is implemented by `AnimalBrainNPC` and its various sub classes. We only need the AI to make the animals move around so that they look alive. The AI controlled animals roam bound to an origin point. Whenever they are too far away from the origin point the `desiredHeading` is recalculated to bring the animal back inside its roaming area. `desiredHeading` is first set to point directly at the origin point. Then a random rotation is applied to the `desiredHeading` to prevent the animals from moving back and forth along the same line which would make them too predictable. The `desiredSpeed` of NPC animals is set to an initial value and is never modified because the NPCs should never stop roaming.

The animal AI also features simple obstacle avoidance. If the animal collides with anything the `desiredHeading` is multiplied by -1 to turn them around. This prevents the player from seeing animals endlessly walking into obstacles.

Land animal AI never invoke any special actions because it is not needed for traversing the terrain. The water animal AI have to invoke their special action when they are stranded on land. The only way for a water animal to move on land is by jumping, so the `WaterAnimalBrainNPC` invokes the special action whenever the water animal is grounded.

The AI for the air animals is slightly more advanced because it features a life cycle. We want them to alternate between flying and walking on the ground. The current life cycle state of the air animals is given by a boolean flag `flying`. When `flying` is false the air animal NPCs function the same way as the other animals. When it is true the air animals will try to stay airborne, this is done by invoking the `ascend` action whenever the animal is not in the air. `ascend` will launch the animal for flight followed by a vertical climb to gain altitude. The `flying` flag is toggled at random intervals to make the air animals alternate between the two states.

Player Brains

The behaviour produced by player brains comes from input given by the player. The `desiredSpeed` is set by pressing the WASD keys, and increased if also holding L-SHIFT. If none of the WASD keys are held then the `desiredSpeed` is 0, stopping animal movement. The `desiredHeading` gets rotated by mouse motion and the ASD keys, so that the player can turn the animals using the mouse. The ASD keys apply a set rotation to the `desiredHeading` acting as an offset to the rotation applied by the mouse. Water and air animals are also affected by the C and space keys, which applies a vertical offset. This allows the player to move in a direction not directly parallel to the camera, because the camera rotation is also set by mouse motion. The `AnimalBrainPlayer` invoke the special actions when the player presses the space key.

6.6.4 Animating Animals

The animal animation system consist of two parts; *forward kinematics* and *inverse kinematics*. It works by rotating the bones(see Chapter 6.2.7) in the animal skeleton to move the limbs of the animals. Forward kinematics is the act of setting the rotations of the bones in a limb. Inverse kinematics is the act of calculating the rotations of the bones such that the end point of the limb reaches a specific target. We use a combination of these techniques to animate the animals, which is an idea based on a GDC talk given by

David Rosen [30].

Forward Kinematics

Animal animations are primarily defined by forwards kinematics. We use it to move the limbs of the animals, such as a leg or the neck. We have a class `AnimalAnimation` that contains a list of `BoneKeyFrames`. An instance of `AnimalAnimation` is a complete forward kinematics animation for an animal. We define animations on a per bone basis, which is what `BoneKeyFrames` is for. An animation is a collection of key frames, a key frame is the state of a bone at a certain point in time. The state of a bone is described by 3 `Vector3`; rotation, position and scale. Rotation is the part of the bone that we primarily manipulate for animations.

By adding key frames to a `BoneKeyFrames`, and adding `BoneKeyFrames` to an `AnimalAnimation` we can build a full animation for an animal. To create motion from a set of key frames we use interpolation over time. Assume we have a `BoneKeyFrames` with an array of 3 rotation key frames: `rotations`. To animate those key frames we interpolate between them using time as the interpolation factor. The `BoneKeyFrames` keeps track of what the `current` active key frame index is, and does the interpolation by interpolating between `current` and `(current + 1) % rotations.Length`. The modulus in the second term makes it so that the animations loop seamlessly, by interpolating from the last key frame to the first.

`BoneKeyFrames` also has a feature we call `KeyFrameTriggers`. You can pass an array of function delegates to the `BoneKeyFrames`, the functions will then be called whenever the `BoneKeyFrames` gets a new current key frame. This makes it so that we can sync logic with the animation without knowing anything about how much time the animation needs to loop. We used it to sync movement sounds with the animations (see Chapter 6.8.3 to read more about audio for animals).

Interpolation is also used to transition between two complete animations. This could for instance be when an air animal transitions from flying to walking on the ground. To interpolate between two `AnimalAnimations`, we play both animations and interpolate between the bone states using time as the interpolation factor. This gives a gradual transition between two animations.

`AnimalAnimation` can take the speed of the animal as input. The speed is used to influence how quickly the animation plays. This makes it so that a walking animation becomes a running animation automatically and fluently as the animal speeds up.

Inverse Kinematics

Inverse kinematics is used to polish the animations produced by the forwards kinematics. The issue with the forward kinematic animations is that they are entirely static and without interaction with the environment. A forward kinematic walking animation for instance is not guaranteed to actually place the feet of the animal at the ground. Since inverse kinematics calculate the rotations of the bones so that the limb reaches a specific target we can use it to ground the legs of the animal. See Figure 23 for a comparison of animations with and without the inverse kinematics polish.

The inverse kinematics algorithm we use is called cyclic coordinate descent(CCD) [31]. There are alternatives to CCD such as using an analytical solution. With analytical solutions you solve the problem of finding the bone rotations using trigonometry. With CCD

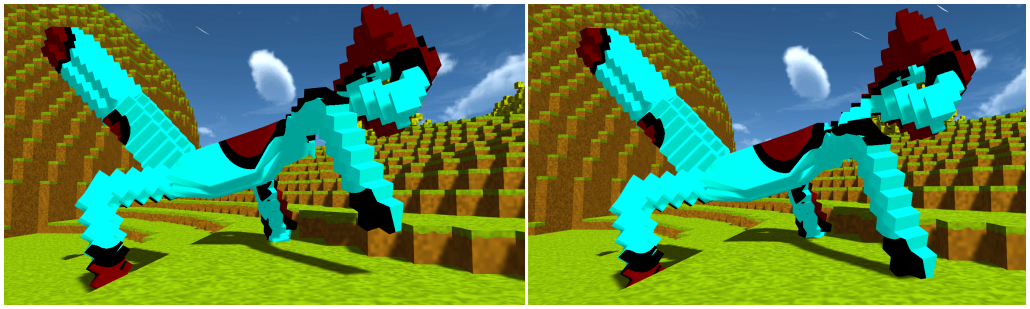


Figure 23: From left to right: Animation without inverse kinematics polish and animation with the polish. The limbs are grounded in the latter image and not in the first.

you iteratively find the angles by rotating the bones towards the target. The benefit of the analytical solution is that it finds the solution in one iteration, whereas CCD could need multiple iterations to complete. The downside of the analytical approach is that it quickly becomes hard to implement as you increase the amount of bones in a limb. CCD can solve for any number of bones, and is easy to implement. The versatility of CCD is what made us favour it over the alternative, because we want the freedom to generate animals with any number of bones in a limb.

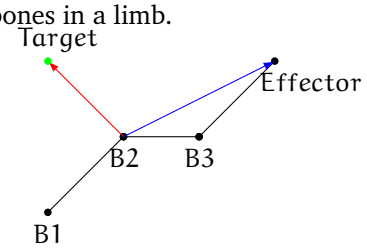


Figure 24: Shows the vectors used for an iteration of CCD on B2. Rotating B2 with the angle between the vectors moves the `effector` closer to the target.

Figure 24 show an image of the CCD algorithm. CCD tries to move the `effector` of the limb to a given target. The `effector` is the end point of the limb. One iteration of CCD applies rotations to the bones of the limb in reverse order, for Figure 24 this would be B3− > B2− > B1. The angle to rotate by is calculated from the angle between the red and blue vectors in Figure 24. To get the direction of rotation we use the cross product of the red and blue vectors. Rotating the bone about the cross product using the angle between the vectors will move the `effector` closer to the target. This process is repeated on every bone until the `effector` is considered close enough to the target.

We use CCD to ground the legs of animals as they walk. We set the ground under the feet of the legs as the target, and the feet as the `effector`. To find the target point on the ground we use the ray cast provided by `VoxelPhysics`(see Chapter 6.5). By casting a ray from the feet and down we find the surface point below the feet.

We also use CCD to make the animals less rigid, we want the tail of animals for instance to be affected by motion. If the animal turns around the tail should look as if it has some inertia. To simulate this we make the tail lag behind the rest of the animal. We use one iteration of CCD with the tail as the limb and the previous position of the tail as the target. This makes the tail lag behind when the animal is rotating or moving in any fashion. The reason for only using one iteration of CCD when simulating limb inertia is

that we do not need to reach the targets, moving in their direction is sufficient.

6.7 User Interface

Because we want the world to be in focus at all times, even when in the menu, we try to keep the UI out of the center of the screen. This is why we decided to keep button placement at the bottom of the screen, as can be seen in Figure 25. This is also the case in the main menu, where we show off a preview of a generated world.

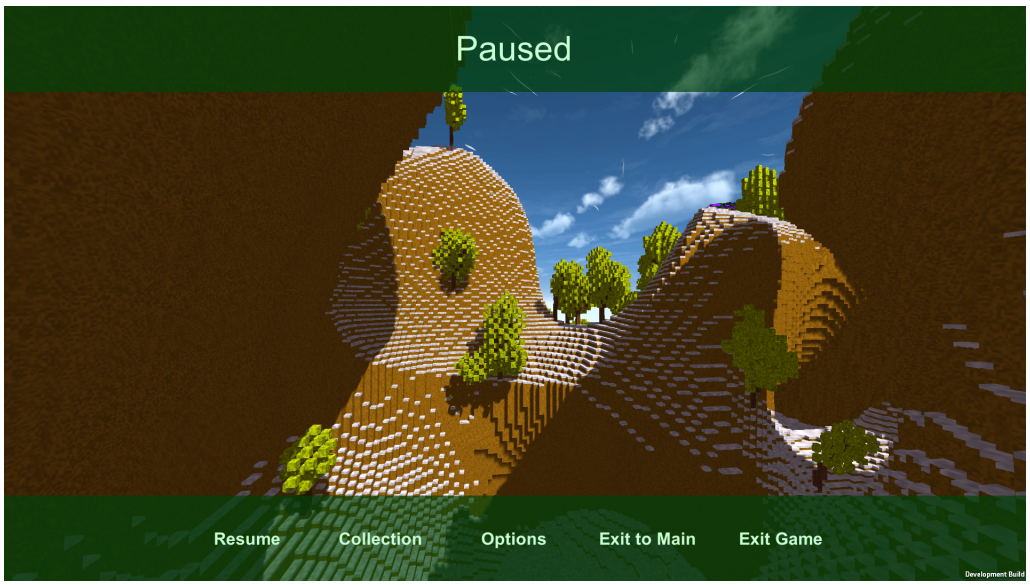


Figure 25: In-game menu UI

The button bar is consistent across all screens, with only the buttons on the bar changing based on the context. This is done by having the different button-sets as children of the button-bar and only enabling them when the context is correct, as can be seen in Figures 25, 28 and 29. This can also be seen in Figure 26, where the button-bar has been marked with the number 3. Just like the button bar, the title bar, labeled 1 in Figure 26 is context based. The item labeled 2 in the figure is the preview window, this is different between the Main Menu and In-game Menu. In the Main Menu this just shows a preview of some world, while in-game, this shows a the world as it was when you paused it. As the title and button-bars are semi-transparent, the preview window is visible behind them as well.

There are 4 button-sets, one for each UI context, where two of the sets are used both in the main menu and in game. The menu contexts are as follows:

- **Base Menu:** This contains the "main" menu context for both the main menu and in-game menu.
- **Settings Menu:** The settings are available in both in-game and in main menu.
- **Animal Collection:** A menu available only in-game, where the player can browse the animals they have collected during the playthrough.
- **Play menu:** A main menu sub-menu where the player can select a custom seed before starting a playthrough.

6.7.1 Main Menu

The Main Menu contains 3 buttons, all in the button-bar, "Play", "Options" and "Exit". These buttons do what you expect, "exit" exits the game, "options" opens the settings and "play" opens up the play sub-menu. The play sub-menu (see Figure 27) changes the context of the button-bar.

Figure 26: Main Menu UI

The play sub-menu contains two buttons, "Start" and "Back" which both do what is expected, either starting the game or going back to the main menu. The sub-menu also has a input field that shows up just above the button bar, where the player can enter in a custom seed if they want, otherwise they will be given a randomly generated seed when pressing "Start".

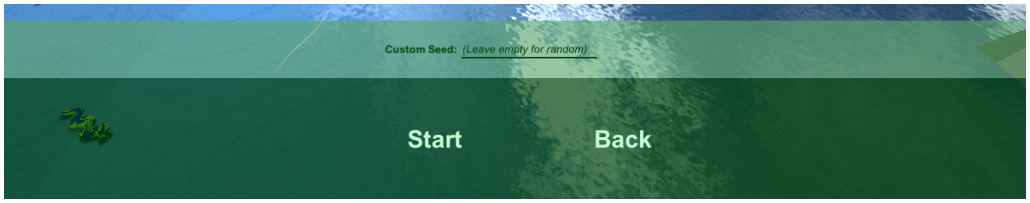


Figure 27: "Play" sub-menu. The input field for custom seed can be seen above the button-bar.

6.7.2 In-game Menu

The in-game menu(see Figure 25) is accessible by pressing the escape-key when in-game, while it is active, the game freezes, this frozen image is what is shown in the preview window. To leave the menu the player can click "Resume" or just press the escape-key again. In the in-game menu, the player has 5 buttons available, including the "Resume" button. The "Options", "Exit to Main" and "Exit Game" do as expected, while the "Collection" button opens up the Animal Collection UI, where they can look at all the animals they have collected (see Chapter 6.7.4).

6.7.3 Settings UI

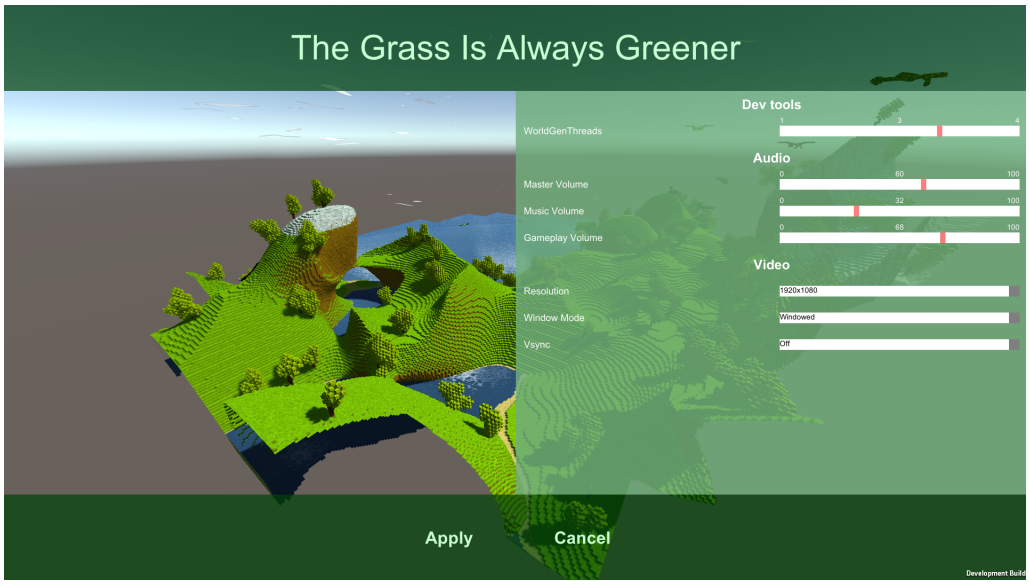


Figure 28: Settings UI

The UI for the settings is generated when the player launches the game. This was done so that it would be very easy for anyone to add new settings to the menu without having to fiddle with Unity's UI tools. The Settings are composed of sections each containing their own settings (eg. video settings, audio settings).

Generating a Section

A settings-section is a container holding one or more settings that the user can interact with. All UI elements in the settings UI are based on what we call a baseUIObject. The baseUIObject is a Unity GameObject containing a RectTransform and a CanvasRenderer, the two components necessary for any UI element. The CanvasRenderer is used by the engine to render the element, and the RectTransform is used to control the location and size of a UI element on the canvas. Sections contain two of these baseUIObjects, a title and a panel. The title contains a piece of custom text that should describe what the section contains, for example "Video Settings" and has the panel set as it's parent. The panel is where the different settings in the section will go. The height and y-position of the panel is set during the building phase.

Generating a Setting

We have 3 different types of settings implemented, these are sliders, dropdowns and input field. They all share a common base called a `basicOption`. The `basicOption` contains 3 parts. Two of the parts are shared among all setting types, the `optionWrapper` which contains the two other, and the `optionText` which contains the name of the setting, eg. "Resolution". The last part is what we call the `interactiveElement`, this is custom for each of the setting types. What the `interactiveElement` is can be discerned from their names. For example, slider settings has a slider as its interactive element. Listing 6.19 shows an example of how we add a new dropdown setting to the settings UI.

Listing 6.19: How a setting is added to the Settings UI

```
GameObject videoSettings = addSection("Video", panel);

string[] resolutions = new string[] { "2560x1440", "1920x1080",
                                      "1280x720", "1024x768" };
GameObject resolution = addDropdownOption(
    optionName: "Resolution",
    parent: videoSettings,
    elements: resolutions,
    saveFunc: delegate { // Update screen resolution
        string dimString = PlayerPrefs.GetInt("Resolution");
        string[] dimensions =  resolutions[dimString].Split('x');
        Screen.SetResolution(Int32.Parse(dimensions[0]),
                             Int32.Parse(dimensions[1]),
                             Screen.fullScreen);
        return null;
    }
);

// ...
```

As Listing 6.19 show, the dropdown takes in 4 parameters. The first parameter, `optionName` is a string used as a key when saving the setting to `PlayerPrefs`. `PlayerPrefs` is a feature built into Unity which allows for saving user settings in between sessions. The second is the parent object, which in this case was the "videoSettings" panel. The third parameter is the elements to fill the dropdown with. For sliders this is replaced by the three parameters `minval`, `maxval` and `isInt`. `minval` and `maxval` is used to limit set the minimum and maximum value, while `isInt` is used to set whether to only allow integer values. With input fields we replace the elements parameter with a placeholder-text parameter. The final parameter `saveFunc`, allows us to add a custom function to run whenever this setting is saved. In our example, the function is used to set the resolution of the game window.

Building the Settings UI

Because there are no rules as to what order sections and settings are added, we cannot predict the positioning or height of a panel when it is created. So once all the sections and their settings have been created, we go through and update their y-positions and heights so that they stack nicely underneath each other.

6.7.4 Animal Collection Display

The Animal Collection Display (seen in Figure 29) is a way for the player to look at the animals they have collected while playing. The UI for the animal collection contains 2 main parts, the animal preview and the animal count. The animal count in the top left shows how many animals you have of each animal type, as well as the total amount of animals. The player can click on any of the types to filter what animal types appear when they browse the animal preview. The animal preview allows for the player to browse through the animals they have collected by clicking the arrows on the sides. In the center of the preview the animals get displayed. The display works by showing the animals from a list `displayCollection` which at all times contain the animals that match the types that the filter allows to show. The display works by putting an animal to display in front of a camera that renders to a texture. This texture is then displayed as an image in

Figure 29: UI for animal collection

the UI. The animal to display is generated using the data stored in a `CollectedAnimal` object. See Chapter 6.10.3 for more on how the animal collection system works, and for an explanation of what a `CollectedAnimal` is.

6.8 Audio

The audio in the game can be separated into three categories: *music*, *environment*, and *animals*. The animals are for the most part controlled by their own scripts (see Chapter 6.8.3), while *music* and *environment* are controlled by the `AudioManager`.

6.8.1 Music

The *music* is controlled by the `musicPlayer()` co-routine. The music player starts playing a track, and then sleeps for the duration of the track, before it plays the next one.

6.8.2 Environment

The 3 environment sounds (wind, ocean, water) are played by the `environmentPlayer()` co-routine, and the tracks for these sounds are all playing on a loop. The environment sounds requires custom checks against the world to update the volume levels, because you can't place the audio source at one single location in the world as they come from multiple places. These checks are ran every frame to keep the environment volume correct.

Wind Audio

Updating the volume of the wind is fairly straight forward, as opposed to updating the volume of the water. We start by finding out how far away from the global wind ceiling we are, and set that as the basis of our search. We can then look for wind-chunks that are closer than the global wind ceiling. We then use the distance from the closest wind-chunk, or the distance from the wind ceiling if no wind-chunks were found to be closer. Wind-chunks and the global wind ceiling is explained in detail in Chapter 6.10.2.

Water Audio

Doing the audio for the water right, was a bit of a challenge. As we had to calculate how far the player was from a point of water every frame, we needed an efficient search method. Originally we tried looking up the BlockTypes of all the blocks in the range `waterSoundRange` of the player, where the range was how far away the water could be heard. This method proved very expensive as it lead to checking the distances from 1 million blocks just with a radius of 50. We tried limiting the search so that it would only check blocks whose y-position could actually be water. This lowered the amount of blocks checked significantly, but was still far from where we wanted.

We eventually came up with a solution where we used the vertices in the water meshes for the chunks. We did this by finding the chunks within a range of `waterSoundRange + Mathf.Sqrt(chunkWidth * chunkWidth * 2)` and sorted them based on how far they were from the player. The reason for the extended range, is because while the chunk's position might have been out of bounds, it might still have contained vertices that were inside in range. Once we have the chunks we want to check, we search through the vertices of the water-meshes of these chunks. By doing this we could find what vertex was closest to the player and use this for controlling the volume of the water.

While the vertex search method was a lot better than searching through the blocks, it was still causing a bit of lag. To increase the efficiency even more, we made a few adjustments. The first one was that we used the distance to closest ocean if our distance from the ocean was less than `waterSoundRange`. This is because the method we use for finding the ocean is a lot cheaper than the vertex search, but also less accurate. The second adjustment we did was that before we started searching the vertices of a chunk, we checked that they were no further away from the player than `closesetVertDist + Mathf.Sqrt(chunkWidth * chunkWidth * 2) * 0.5f`, where `closesetVertDist` was the the distance from the closest vertex we had found. This is because if they are further away, then it is not possible that they contain any vertices closer than the currently closest vertex. If a chunks was to far away, we skipped it and went on to check the next chunk. When going through the vertices of a water mesh, we only check every 4th vertex as a face/quad contains 4 vertices and we only need to check one vertex in each face.

Ocean Audio

Because of the size of the ocean, we found that we could use a less accurate method to find ocean water, than we could to find smaller lakes that weren't part of the ocean. Having the ocean checking separate also allowed us to have extra audio in the ocean water. The distance from the ocean is used to control the volume of ocean water.

Finding the ocean water depends on the fact that we only add wind areas in the ocean biomes. We start off by finding the closest wind-chunk (see Chapter 6.10.2) and calculating our (x,z) distance from it. If we find a wind-chunk that is in range, we add the distance in the y-direction into the calculation. Calculating the player's distance from the water is done in two ways, depending on whether the player is above or below the corruption water offset (see Chapter 6.2.4). If above, we find the distance between the player's y-position and `waterLevel + corruptionOffset`, while if the player is below, then it is calculated as the distance between the player's y-position and `corruptionOffset`. While calculating the distance from the ocean like this is not

very accurate, it is accurate enough for this use and very efficient.

6.8.3 Animal Sounds

All animals make sound, they "speak", they make sounds when moving, and they make sound when interacting with certain parts of the environment. The animals' audio is controlled by the `AnimalAudio` script, this script manages the `AudioSource` attached to the animal, controlling volume, pitch and what is being played.

Speech

Their "speech" is controlled by a co-routine, this co-routine sleeps for 10 to 20 seconds at a time, and plays the "speaking" sound when awake. The pitch of the sound varies slightly each time it is played, to create some variety in the sounds the animals make.

Movement Sound

The playing of movement is not entirely in the control of the `AnimalAudio` script. The timing for playing these sounds is determined by the animation. When we set up the animations' key frames, we can add in what we call KeyFrameTriggers, functions that are run when a key frame in the animation is completed. More on animations and on how the `KeyFrameTriggers` work can be found in Chapter 6.6.1. We add in callbacks to functions in the `AnimalAudio` script that deal with playing the different movement sounds.

All animals have a walking sound that they make when traversing on ground, the sound made can vary depending on what surface the player is walking on. What surface the player is walking on, is found by checking the type of the block below the player using our custom `VoxelPhysics` class (see Chapter 6.5). The movement sound is put into a `KeyFrameTrigger` for the walking-animation of one leg on the land animals and air animals. The flying animals also have the sound of wing flapping when they are in the air, this is put into a `KeyFrameTrigger` for the flying-animation of one of the wings. The reasoning for only putting the sound triggers into one limb, instead of into all the relevant limbs, is that when these sounds are all played (almost) simultaneously from the same `AudioSource`, it creates artifacts in the audio that we do not want.

Water Splashing

The animals also have audio for some interactions with the environment. When an animal hits the water-surface a splashing sound is made. Due to the fact that water animals interact with water differently than air and land animals, there are two ways for the splashing sound to be made. The first way, used by water animals plays a sound when the animal enters the water. As we use a custom collision-detection system, how entry is detected is explained in Chapter 6.5. The second way, used by land and air animals, is necessary because they cannot enter into the water in the same way as water animals, but instead stay floating on top of it. This method works by checking the animal's state (see Chapter 6.6.1) to see if it is sitting on the water surface. If it is sitting on the water surface, but was not doing so last frame, then we know that they just entered the water and can play the sound.

6.9 Shaders

The textures on the terrain, trees and animals are generated at run time on the GPU using shaders that we wrote. By using the GPU for this we leverage the massive parallel computing power that GPUs have leaving the CPU free to do other things, which is good for performance. The game world uses 4 main shaders; Terrain.shader, Tree.shader, Water.shader and Animal.shader. All of the shaders share the same core logic for generating the textures on the mesh they render, they take some input data from the actual mesh and combine it with noise to output a color value for the surface of the mesh.

The normal method of doing textures is to create the textures in advance then loading them as resources for the GPU. The GPU then samples color values from the texture and uses it to color the mesh it is rendering. Our method of coloring the meshes differs from the traditional method in that we generate the color at run time using noise as mentioned above. Doing this gives us a more varied look on the things that we render, as well as making the textures seamless since our noise function has an infinite domain as opposed to textures which are finite in size. See Figure 30 for comparison of finite and infinite textures. The border tiling issues seen in the finite textures can be solved by making the edges of the textures equal each other.

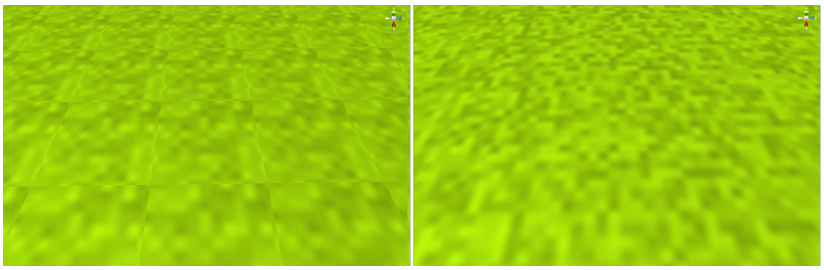


Figure 30: Comparison of finite(left) and infinite(right) textures

Before explaining how our shaders work we will briefly discuss how shaders work in general. A shader is a program that runs on the GPU, the purpose of a shader is rendering a mesh to the monitor. A mesh is a collection of vertices (points in space), these vertices are the first thing that the shader program processes. The shader program runs in a pipeline, with multiple steps that work to convert a mesh into pixels on the monitor. The shaders we write implement the vertex and fragment shader steps.

The vertex shader step of the shader pipeline takes the vertices from the mesh and does calculations for each vertex in isolation and in parallel. So in any given execution of the vertex shader you will only have the information of a single vertex, and all of the data belonging to that vertex. The data belonging to a vertex could be its position, the normal, color, uvs and more. The calculations that are done in the vertex shader is usually something like transforming the vertex, but you can do a lot more. Once the vertex shader has completed, it passes on its results to the next step in the pipeline which in our case is the fragment shader.

A fragment is a single pixel sized piece of the mesh and the output of the fragment shader will contribute either nothing, partially or fully to the final pixel value on the monitor. The input to the fragment shader is a composite value of potentially multiple vertices that have gone through the vertex shader. This composite value is calculated by a

rasterization step in the shader which happens before the fragment shader. Rasterization interpolates the values of multiple vertices to produce a single fragment, this is needed because a fragment could be in the middle of a face defined by multiple vertices. The fragment produced by the rasterization is then input to the fragment shader. The job of the fragment shader is primarily taking the input fragment and calculating its final color.

We will mostly be explaining the parts of our shader implementations that relate to generation of textures and use of noise functions. Some key things we do not mention here is: the spatial transformations applied to vertices and fragments using the MVP matrix, lighting (ambient, diffuse and specular) and shadow calculations.

6.9.1 Terrain shader

The terrain shader is tasked with rendering the mesh for a chunk of terrain to the screen, see Figure 5 for an image of a chunk. There is one main challenge associated with rendering a chunk, which is the fact that different blocks should have different textures and one chunk mesh can contain any number of different blocks. This means that the shader needs a way to tell what kind of block it is currently rendering. Once the shader has obtained the block type of the vertex it then needs to know what color that block type should be.

The normal data associated with a vertex is not fit for directly representing a block type so we had to get creative when choosing how we would do it. We chose to encode the block type represented as an int into the vertex color data when we create the mesh for the chunk. We put the int representation of the block type into one of the color components. When we retrieve the block type from within the shader we have to add a small number to the block type before casting it to an int due to floating point precision limitations.

Now that we can get the block type of the vertex/fragment by decoding it from the vertex/fragment color data we need to figure out what color that block type should be. To solve this problem we declared 3 static arrays in the terrain shader containing data used for generating the final color of the fragment, the block type int is used to index these arrays. See Listing 6.20 for the declaration of the arrays.

Listing 6.20: Terrain shader arrays

```
static const int COLOR_COUNT = 5;
static float frequencies[COLOR_COUNT] = {
        7.74,   //Dirt
    //...
};
static fixed3 colors1[COLOR_COUNT] = {
        fixed3(0.729, 0.505, 0.070),    //Dirt
    //...
};
static fixed3 colors2[COLOR_COUNT] = {
        colors1[0] / 1.5,       //Dirt
    //...
};
```

The frequencies array is used to get the noise sampling frequency for the block type, and the two color arrays gives the two colors of the block type. To calculate the color of the fragment we sample the noise value using the fragment world position and the frequency of the block type. Once we have the noise value for the fragment which is in the 0 to 1 range we use it to interpolate between the two block type colors. The

interpolated color is the final color value for that block type. To read more about the noise function see Chapter 6.1

However, blocks can have modifiers giving them two block types, the block seen in Figure 5 is a dirt block with grass as a modifier. We can get the modifier block type and color the same way we get the normal block type and color, the issue now becomes determining which of the two block types to use for the final color. We can solve this issue by using the uv of the fragment, a uv is a texture coordinate which ranges from 0 to 1 in the x and y axis. The uv for our blocks start at `uv.y = 0` at the base of the block and increases to `uv.y = 1` at the top of the block. Because of this we can look at the y value of our uv to determine if we should use the block modifier color or the block base color for coloring the fragment. When the y value of the uv is greater then 0.8 we use the block modifier, when its lower we use the block base for coloring the fragment. We also add some noise to the 0.8 threshold to prevent the base/modifier transition from being a straight line.

Block modifiers do cause some rendering artifacts when rendering them at a distance. The artifacts can not be captured by a still image, they appear as waves of aliasing which looks like an interference pattern in the terrain when the camera is moved in game. It is caused by the fact that distant blocks with modifiers get few pixels to represent them, this combined with limited precision causes the fragments to switch between the block base and modifier color quickly when moving the camera.

The problem we encountered is usually solved by using a mipmap [32] when using normal textures. A mipmap is a set of the same texture at decreasing resolution, when rendering distant objects you would use a low resolution texture to decrease the detail level. The process of decreasing detail level as a function of distance is known as LOD(level of detail) [33].

Since we are not using textures but generating them using noise we can not use mipmaps to solve our problem. We decrease the detail level from the shader code instead. To decrease the detail level of block modifiers we gradually move the 0.8 modifier threshold mentioned earlier to 0 as the distance from the fragment to the camera increases. This makes distant blocks turn into their modifier entirely, decreasing the complexity of distant blocks by reducing them to a single color. We also reduce the noise sampling frequency of distant fragments which gives the textures fewer large scale features. The reduction in frequency at a distance also helps alleviate graphical artifacts and is analogous to mipmapping. See Figure 31 for an up close comparison of the two levels of detail.

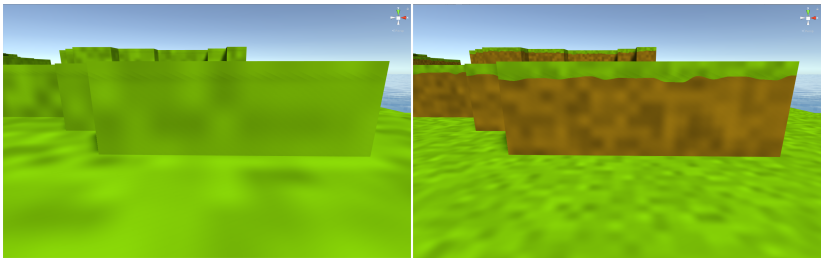


Figure 31: The image shows from left to right: A chunk with a low LOD (low noise frequency) and the same chunk with a high LOD (high noise frequency)

6.9.2 Tree shader

Just like it is the terrain shaders job to render chunks of the terrain the tree shader renders trees, see Figure 18 for an image of trees. The tree shader has to solve the same problems that the chunk shader does, which is determening block type and color and it implements the solutions in the same manner. The blocks for the trees are wood and leaf, so the arrays in the tree shader are only 2 items long. Trees does not have modifiers for their blocks so the terrain and tree shaders differ in this way, this also means that trees do not need the level of detail system used by terrain either.

There is one difference in how trees generate color, when the block type is wood the color is not always an interpolation between the two colors of wood. Instead it is either fully one of the colors when the noise value for the fragment is below or above certain thresholds or it is interpolated as usual when the noise value is between these thresholds. This gives the texture of wood more sudden transitions between the two colors of wood, giving the wood a bark-like look as seen in Figure 32. You can compare the sudden transitions to the normal interpolation by looking at the leaves which are always calculated as the interpolated value of the two colors using the noise as the interpolation factor.

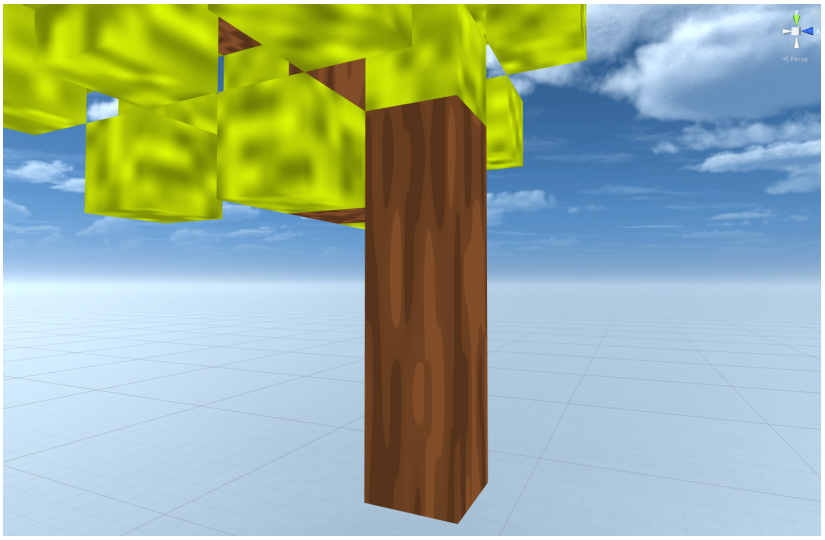


Figure 32: Image of wood texture

6.9.3 Water shader

There are two main effects that we want to accomplish with the water shader, we want the water to be semi transparent and reflective. Achieving transparency is easy, we just have to set the alpha value of the color for the water fragments to some value less then 1. The reflective property is also easy in unity, we can use macros provided by the engine to reflect the sky. The sky in our game as in most games is implemented as a skybox, the skybox is textured by a cubemap image. The reflection of the sky in water is implemented by sampling the cubemap for the skybox and using the color values of the cubemap as the color values for the water fragments. We use a unity macro for doing the cubemap sampling, the macro needs the normal of the water surface to work out where to sample the cubemap. When we provide the macro with the normal we get color value for the

reflection, setting the sampled color as the water fragment color gives us water that reflects the sky.

However, only reflecting the sky in this way creates a very strange mirror like water, the water looks like a perfectly flat big mirror which is unnatural. Real water has an uneven surface with constant motion which is the look we want for our water as well. The first thing we tried was manipulating the vertices of the water using noise, so that the water surface would become uneven. Changing the actual geometry of our water in this way caused issues with other things such as lighting and shadow calculations. We instead used a slightly less obvious approach where we used noise to slightly alter the water normals. This simulates an uneven surface without the surface actually being uneven, because we now have normals that point in slightly different directions as they would if the surface truly was uneven. We also use the amount of passed time as a factor when we sample noise to simulate motion on the water surface. See Figure 33 for a comparison between the flat and uneven water surface.

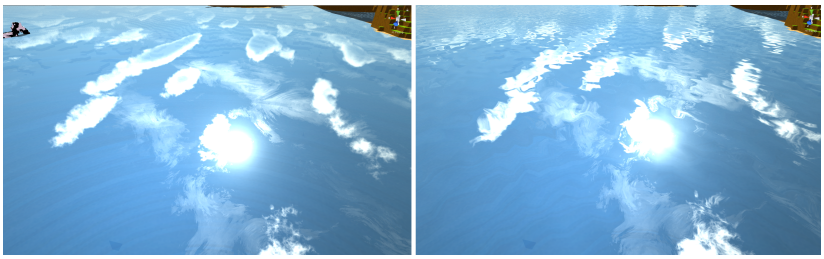


Figure 33: the left image shows water with unaltered normals, the image on the right is water with noise applied to the normals.

6.9.4 Animal shader

We have been very liberal with how we texture the animals as you can see in Figure 19 and Figure 20. They can have a virtually infinite combination of two colors in all sorts of patterns. The animal shaders differ from the terrain and tree shaders in one fundamental way, instead of storing and looking up colors in arrays the animals use noise to generate the colors themselves.

We still use two colors as with terrain and trees, and we still transition between them using noise. The transition itself is not done through using the noise to interpolate, for animals the transition is instant once the noise value crosses a threshold. The threshold for when to use color1 or color2 has a gap in it, the gap is filled with nothing producing a black color in what would have been the transition between the two generated colors. The black gap was initially introduced to fix a graphical glitch caused by the transition but we kept it even after working out how to solve the glitch because we liked it.

The implementation of the animal shader had 2 main problems for us to solve. The first problem is consistent noise sampling, we use the fragment world position to sample noise which causes the texture of the animal to move when the animals moves. The second problem is consistent generation of colors for the animal, every fragment has to be computed independently and we want all of them to generate the same two colors for the animal.

We solved both of these issues with an approach similar to what we did for terrain

and trees, by encoding some data into the mesh of the animal. The noise sampling issue was solved by storing the default vertex positions of the animal mesh into the vertex color data. As mentioned in Chapter 6.9.1 the vertex color data is a collection of 4 floats, so to store the original vertex positions we use the first 3 floats for the vertex x, y and z coordinates. This gives us static positions for every vertex that does not change as the vertex is transformed, solving the sampling problem.

In order to make the color generation consistent across all fragments we put data used for noise sampling in the uv(texture coordinate) of the vertex. We put a seed into the y component and a frequency into the x component. We use the uv seed and frequency for more then just noise sampling, we also use various compositions of them to generate the colors for each fragment. The uv data is a good candidate for this task because it is consistent across all fragments in an animal(it is not generally the case that the uv is the same for every vertex, they are the same for animals because we made them all the same). The colors that the uv is used to generate is a collection of 3 floats in the 0 to 1 range and we want to generate a large range of colors. We want a small change in the uv to result in a large change in color, to give the animals as much variety in texture as possible. These requirements resulted in the code for generating colors seen in Listing 6.21, the animalData object is the uv.

Listing 6.21: Animal shader color generation

```
half3 color1 = {
    cos((i.animalData.x * i.animalData.y)  * 517.72), //R
    cos((i.animalData.x + i.animalData.y) * 444.54),  //G
    sin((i.animalData.x / i.animalData.y) * 314.22) };//B
//color2 is similar, but not numerically the same
```

Some of the things that the color generation in the Listing 6.21 accomplishes is that it ensures a different value for the R, G and B components even though they are calculated from the same inputs by using different operations. It ensures a good spread of colors by multiplying them by a large number. The use of trigonometric functions sets the final value in the -1 to 1 range, the reason for wanting negative values is the ability to completely disregard a component. The final color values gets clamped to the 0 to 1 range, so negative values become 0. The motivation for dropping one or multiple components of the RGB color is that we do not want every animal to have a composite color which would be the case if we generated RGB values in the 0 to 1 range. Blue and yellow animals such as the one in Figure 20 would not be possible without dropping RGB components, blue is composed of only one value, and yellow is a composite of green and red. It is numerically possible for a color generated with components in the 0 to 1 range to also have 0 or near 0 components but it is far rarer compared to our solution, which is a conclusion we arrived at through empirical data after we tested both solutions. There is one pitfall to generating colors in the -1 to 1 range as we do which is when all components of the color gets dropped producing all back animals. To combat pitch black animals we check if both colors are black and make one of them white, this event is rare so you will rarely see a black/white animal while playing.

6.10 Gameplay

In this section we will talk about different gameplay elements that we have in the game. To keep the game's focus on the world around the player, we decided to focus the game-

play around using the different animal types so that the player can see and explore the world in multiple ways.

6.10.1 Animal Switching

The animal switching allows for the player to switch body with the animals they find in the world. To take over another animal, the player has to face the animal and press down the left mouse button. When the player clicks down the mouse button, we have to try figuring out what animal they wanted to take control over.

To find what animal the player wants to swap brains with, we have to find the best match in a list containing all animals. As animals are stored in pools, we have to go through every animal pool and all the animals in them. For every animal we are checking, we calculate the angle between the camera's forward vector and the vector of `animalTransform - camaraTransform`. If the animal we are checking is close enough, and the angle is small enough we store the index of the pool the animal is stored in, and the animal's index in the pool. If an animal was found, then the player will be given control of the other animal.

When the player takes control of another animal, the brain of both the new and old animal will have to be swapped out. The old animal will be given an NPC brain, and the new animal will be given a Player brain. How the brains work, and how NPC and Player brains differ can be found in Chapter 6.6.

6.10.2 Wind

Wind is something we decided to implement in order to give the player more incentives to try out the different animal types. Without the wind, the player could just find and become an air animal, and then just fly in a straight line to the end of the world. Now they instead have to switch around if they want to traverse the terrain efficiently. If the player enters into a windy area, the force of the wind will push against them when they are in the air, making it near impossible to fly through.

Wind can be found anywhere if the player's y-position is higher than the global wind ceiling of 140, it can also be found in designated wind zones placed around in the world where the wind will affect the player as long as they are not grounded. These wind zones that go all the way down to the ground exist in ocean biomes, but only outside the radius of 100 around (0,0). The reason for not having wind within 100 of (0,0) is that we do not want the player to get stuck in the wind, and also that the visual effect of the wind end doesn't look very good when all the wind is converging into a single point. There are three ways in which the wind is used. The first is as said above, it acts as a force on the player. This is done by changing the player's velocity to add `windDirection * globalWindSpeed`, where `windDirection` is calculated to be in the direction of (0,0), and `globalWindSpeed` is how strong the wind is. The second way the wind is used is audio, how we use the wind for audio is explained in detail in chapter 6.8.2.

The final way in which we use the wind is as a visual effect. The visual effect is created using particle systems. These particle systems are instanced on a per-chunk basis, all chunks outside of the 100 range around (0,0) contain their own particle system. If the chunks closest biome is an ocean biome(see Chapter 6.2.3), the particle system goes from y=0 to y=300. If the biome is not an ocean biome, then the particle system goes

Figure 34: Close up of a wind particle.

from y=140, which is the "global wind ceiling" up to y=300. The reasoning for having the y go as far up as 300, is that we have the world height set at 200 and we wanted to have the wind go above the world height, so that even if the terrain got that high, the player wouldn't see where the wind particles stopped. Unity's particle systems allow for us to add trails to the particles. To get the effect on the particles as seen in Figure 34, we disabled the material on the particle itself, and only used the particle trails. We also use the built in noise functionality in Unity's particle system to add some noise to the movement of the particles, so that the trails don't end up being straight lines. This visual effect also serves as an indicator for which direction the player needs to go in order to get to the end of the world.

6.10.3 Animal Collecting

As the player traverses the world they are likely going to switch animals many times. We wanted the player to be to look back at what they have collected, this is what led to the animal collection system. In this section we will be explaining how the animal collection works behind the scenes. The UI is explained in Chapter 6.7.4.

The animal collection is controlled by the `AnimalCollection` class. The class contains two lists containing objects of type `CollectedAnimal`, one containing all the collected animals called `collectedAnimals`, and one containing only the animals that are to be displayed in the UI called `displayCollection`. A `CollectedAnimal` consists of the `Type` of the animal, and the seed needed to replicate the skeleton and texture.

Whenever the player switches to a new animal, it is automatically added to the animal collection, so that the player can look at it from the collection display at a later time. When we want to add a new animal to the collection, the first thing we do is check that there are no animals in the `collectedAnimals` list that matches the type and seed of the animal we are trying to add before we allow the animal to be added.

6.10.4 Win Condition

To win the game the player has to reach the end of the world. The above game mechanics are tools that the player can use to get to the end. By selecting the best animal for the current environment. When the player reaches the end of the world the UI shows the player that they have won. The UI also gives some statistics about the play-through which

are; time spent reaching the end and distance traveled. See Figure 35 for an image of the win condition being triggered.

Figure 35: Image of victory screen shown when player reaches the end of the world.

6.11 Implementation Statistics

We figured it could be interesting to add some statistics for the implementation.

Lines of Code

- **Lines of `C#` code:** 9748
- **Lines of shader code(**.hlsl and .shader files**):** 1053
- **Total line count:** 10793

With a total of 93 `C#` and shader files, this gives us an average of 116 lines in every file.

Breakdown of source code:

- World Generation: 31%
- Animals: 29%
- Shaders: 10%
- Utils: 10%
- UI: 7%
- Other: 13%

The above data was obtained by counting lines of code inside individual folders of the project.

Hours spent

As very little programming was done after sprint 14, and we did not log time the first week, this is data from sprint 2 through 14. We spent about 500 hours programming the game, with an average of about 39 hours collectively each week.

Breakdown of time usage:

- World Generation: 23%
- Animals: 21%
- Shaders: 8%
- Physics: 6%
- Audio: 5%
- UI: 3%
- Other: 34%

The above data is a rough estimate of how the time was spent. We obtained the data by searching for key words in our time logs. When we log our work we write a description of what we worked on along with the time, such as "Worked on making AirAnimals fly". The other category encompasses anything not directly relating to the other categories. This could be things such as making the benchmarks, fixing bugs and more.

7 Optimization

In this chapter we will cover some of the more interesting and impactful performance optimizations we made during the development of this project.

7.1 Methodology

7.1.1 Benchmarks

To be able to get some solid data on how the game was performing, we made two benchmarks that we used to measure the game's performance. The two benchmarks are called *RealBench* and *SynBench*. Both of these benchmarks allow for us to set what thread-counts we want to test for. See Appendix E for concrete examples of the output data from both benchmarks.

All benchmark results shown in this chapter were obtained by running the benchmarks from a build of the game, as opposed to from within the Unity editor. The benchmarks were also run on the same system, with the following specifications:

- CPU: AMD Ryzen Threadripper 1950X 16/32 cores/threads
- GPU: Nvidia GeForce GTX 1080 Ti
- RAM: 32GB DDR4

RealBench

RealBench is used for measuring the frame rate over a set amount of time while moving in a straight line and generating terrain and animals. It also tracks how many chunks are generated, how many chunks are canceled after having been queued for generation and the number of animals generated. This benchmark allows us to set how many seconds we want to run it for.

RealBench works by creating a dummy player, and feeding it into the `WorldGenManager` (see Chapter 6.3). This dummy player flies through the air above the terrain at the max speed an animal can move so as to represent a somewhat realistic movement speed. While the `WorldGenManager` generates the terrain, we run a timer in the `RealWorldBenchmarkManager` and count the frames until the timer is done. After the benchmark has run the set time we calculate the average FPS, get the data on generated chunks, cancelled chunks and generated animals and write this to a file together with the raw FPS output over time. We can then create a graph using the raw FPS data we recorded. For generating the data for the graphs used in this chapter, we ran RealBench at 8 threads for 4 minutes.

RealBench has been changed somewhat over the course of development. When going back to old commits to get benchmark data for this chapter we manually applied the most recent changes to make the different results comparable. The difference between the old version and the new version is how the dummy player and the camera moves. In the old version of RealBench the player would move in random directions whereas in the current version the player moves in a straight line. We changed it to a straight line to remove the movement pattern as a variable making different benchmark results more

comparable. The camera would rotate around the player in the old version, we removed this in the new version because the rotation caused cycles in the RealBench FPS graph which made the results harder to interpret.

SynBench

SynBench is used to measure the speed of the terrain generation and animal generation by measuring the time used to generate a 20x20 chunks and 20 animals.

Unlike RealBench, SynBench does not make use of `WorldGenManager` to generate the terrain. Instead, it handles chunk and animal generation by itself. The way it differs from `WorldGenManager`(see Chapter 6.3) is that it does not pool objects, as we generate only around (0,0) so no chunks or animals will be unloaded. The `SyntheticBenchmarkManger` also orders all the animals and chunks at the same time just after starting. The process from a chunk is ordered until it is returned to the main thread is the same as with the `WorldGenManager`. While the worker threads are generating the terrain and animals the `SyntheticBenchmarkManger` keeps counting the frames in the same way as RealBench. Before the animals and chunks were ordered a timer was started. This timer keeps going until all the chunks and animals have been generated and launched into the world. We now calculate the average FPS and write this to file together with the time used to generate everything. To generate data for graphs from this, we run it over multiple iterations with different thread-counts. For generating the data for the graphs used in this chapter, we ran SynBench at 1 to 25 threads at a 2 thread interval.

SynBench has one issue in terms of precision. The animals generated by SynBench are generated with random parameters, this becomes an unnecessary variable for the benchmark results. Luckily animals account for a small percentage of the total work so the error caused by this is small. We would have changed SynBench to generate animals with fixed parameters, however this would make the results less comparable with older versions of SynBench which uses random parameters.

7.1.2 Performance Monitoring

While our benchmark systems were nice for testing the overall performance of the game, they don't give much indication as to where eventual performance issues stem from. For this we had some performance monitoring tools that we could use.

Unity has a built in tool for performance profiling called the *Unity Profiler*. With this tool we are able to monitor CPU usage, GPU usage and memory usage so that we can easily see what parts of our code is causing performance issues. The profiler also lets us get an overview of active physics objects and an overview of playing audio sources and how much CPU and memory is used for the audio.

Though the profiler is a very handy tool and it helped us a when optimizing code that runs in the main thread, it does not support multi-threading, so we could not use it to monitor resources used by the worker threads. For this we had to manually do timings using the `StopWatch` class from the `C# System.Diagnostics` library, which allows for easy time-measurements of sections of code.

7.2 BlockDataMap Implementation

7.2.1 Old solution

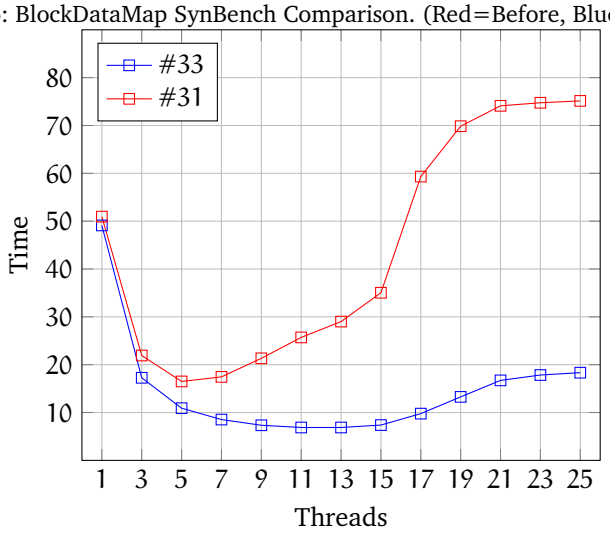
Before we implemented the BlockDataMap as a way of storing the BlockData for the blocks in a chunk, we stored the BlockData in a multi-dimensional array that we were passing around between functions. At this point in time, the BlockData was a class, so the data was pass-by-reference, meaning we weren't making new copies of the BlockData every time it was passed around.

While this solution was very simple, it was more resource intensive then what we wanted. With the BlockData class containing 2 BlockTypes, which is an enum with underlying type `int`, we were using as much memory for the references to the BlockData, as we were using for the BlockData itself. This is important because each chunk at this point contained 20x20x100=40000 blocks, meaning that we were using 40k*16B=640KB of data in each chunk (8B reference + 2*4B int) with only half of this being stored sequentially in memory. This also means that the BlockData itself is stored in random locations in memory, which could potentially mean that to go through all the blocks we would have to read in 40000 cache lines to read in the BlockData from memory.

7.2.2 New solution

The solution we came up with to replace the multi-dimensional array of BlockData was the BlockDataMap, a 1 dimensional array inside a wrapper class. With the BlockDataMap we turned the BlockData class into a struct. That way all the BlockData would be stored sequentially in memory, leading to a lot less cache lines having to be read to load the data from memory. Because the BlockData was now a struct, we also halved the memory it used due to data contained within being the same size as a reference of a 64bit system.

Figure 36: BlockDataMap SynBench Comparison. (Red=Before, Blue=After)



7.2.3 Performance Impact

From the graph in Figure 36 we can see a performance increase across all thread-counts, with the result on high thread-counts being almost 4 times as fast with the new solution. We believe this is due to the improvement in memory layout, going from storing the

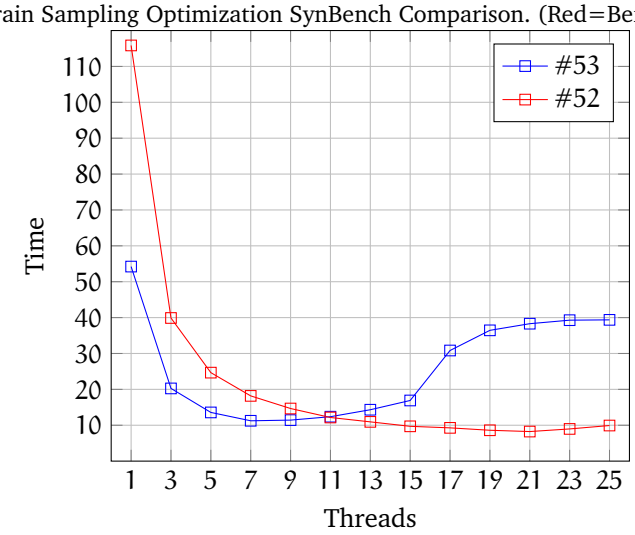
BlockData randomly in memory with sequential references, to removing the references and storing the BlockData itself sequentially.

7.3 Terrain Sampling Optimization

Before the terrain sampling optimization, we had been sampling every block in a chunk with 3D noise. With the chunks being 20x20 and 200 blocks tall, this meant that we were sampling 80 thousand blocks for every chunk we generated. Before this optimization, we had come to the conclusion that there was little more we could do to lower the time needed to sample a single block. Because of this we decided to try look for a way to sample less blocks.

The goal of the terrain sampling optimization was to speed up chunk generation by sampling blocks selectively. Because not all blocks would be sampled with 3D noise, we now also had to pre-initialize the BlockDataMap using data from the heightmap only. Once the map was initialized we added the blocks at (x, heightmap[x,z], z) for every x and z, as well as the blocks at the sides of x and z to a queue for sampling (`samplingQueue`) and to a boolean 3d array to make sure no block was queued more than once (`doneArray`). For each block, we checked whether the sampling with 3D noise gave the same result as the result from heightmap only. If the result differed, we changed the value in the BlockDataMap and tried to add the 6 neighbours to the queue as well, unless they had already been queued or sampled. See Chapter 6.2.2 for a more thorough explanation of the selective sampling.

Figure 37: Terrain Sampling Optimization SynBench Comparison. (Red=Before, Blue=After)



7.3.1 Performance Impact

For low thread-counts the selective sampling show the expected result, less time was needed because less blocks were sampled. For higher thread-counts can see in Figure 7.3 that the selective sampling actually ends up taking more time.

We believe this could be because we no longer read the BlockData sequentially. With the old solution we would just iterate over the BlockDataMap, but with the selective sampling the access is more random which causes worse cache utilization due to random

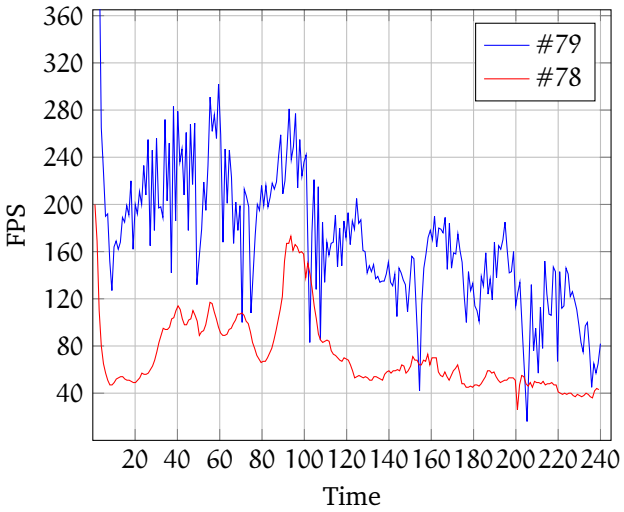
access instead of sequential access. This shows very well with high thread-counts as the threads will be competing for the cache memory.

7.4 Shader Optimization

We generate the textures of everything procedurally at runtime on the GPU using custom shaders. The mesh of a terrain chunk can contain multiple block types, each with their own texture. This means that we need to texture them differently. We encode the block type into the mesh color data, so that we can tell from the shader what block type the current vertex belongs to. In our first approach for procedural textures, we would input the block type to a switch statement with logic for each block type. This worked and produced correct looking results, the performance however had potential for improvement. The issue is that GPUs are bad at branching [34], which a switch case for every block type causes to a large extent.

To solve the branching issue we made new shaders without any branching. The new shaders use the block types to index arrays of data, which is then used to generate the textures. So instead of having separate logic for each block type, we now have the same logic but separate data. To read more about the current shader implementation see Chapter 6.9. In addition to changing the shaders we also changed the chunks themselves. We quadrupled the area of each chunk, the reason for this is that GPUs render one chunk at a time, so having fewer bigger chunks gives us better batching of vertices. This helps us leverage the massive parallel computing power offered by GPUs. The change in chunk size also had a surprising effect on terrain generation performance as seen in Figure 39. We also reduced the amount of chunks in the `WorldGenManager` and SynBench after the change, so that the size of the generated world remained unchanged.

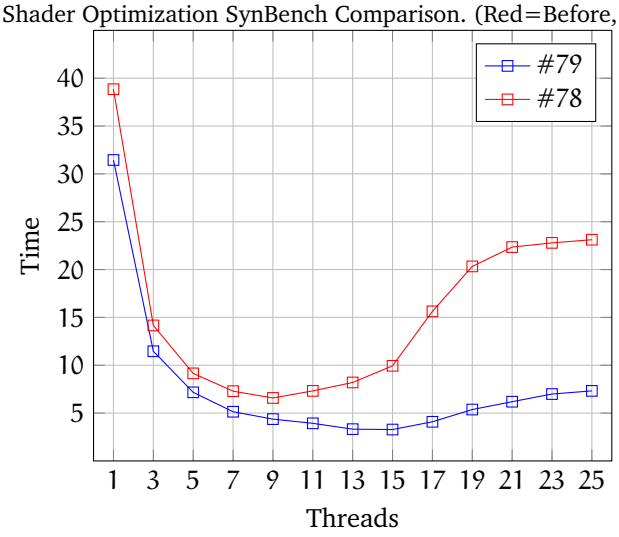
Figure 38: Shader Optimization RealBench Comparison. (Red=Before, Blue=After)



7.4.1 Performance Impact

As seen in Figure 38 the rendering performance increased by a large factor. The lack of branching in the shader and the increased batching of vertices had the desired effect. Average FPS for the two RealBench runs were 72 and 168, which is an increase of 133%.

Figure 39: Shader Optimization SynBench Comparison. (Red=Before, Blue=After)



What we did not expect from these changes was an improved terrain generation speed. We suspected that it was caused by the terrain sampling optimization from Chapter 7.3, with bigger chunks it manages to avoid even more sampling. We confirmed this by checking out a commit from before the terrain sampling optimization, then we made the chunk size change again and ran SynBench. SynBench showed no improvement in score in that case.

Note About Results

In the original pull request for shader optimization we forgot to adjust the spawn rate of animals for the chunk size increase. Before running the benchmark we manually adjusted the animal spawn rate, so that it was at its intended level, which is 4 times higher then it was in the pull request. The animal spawn rate issue was not fixed before pull request #84, and the manual adjustment has been applied to every pull request from this one (#79) to #84 before running benchmarks.

7.5 Physics Optimization

Doing collision detection for the chunks in our game gave us performance issues for a long time. It was not the detection of collisions itself that caused frame-rate issues, rather it was the initialization of the colliders used for the terrain. The frame-rate issues we had was also in a displeasing form, with irregular spiking giving the sensation of stuttering when playing the game.

7.5.1 Mesh Colliders

We used to use mesh colliders for every chunk to enable collision with the terrain. The issue with using mesh colliders was the large overhead of initializing them. The fact that our chunks have a relatively high vertex count makes mesh colliders computationally expensive to initialize as seen in Figure 40.

We tried to remedy the cost of the mesh colliders on the terrain by introducing what we called lazy collisions. Lazy collisions is based on only enabling mesh colliders for

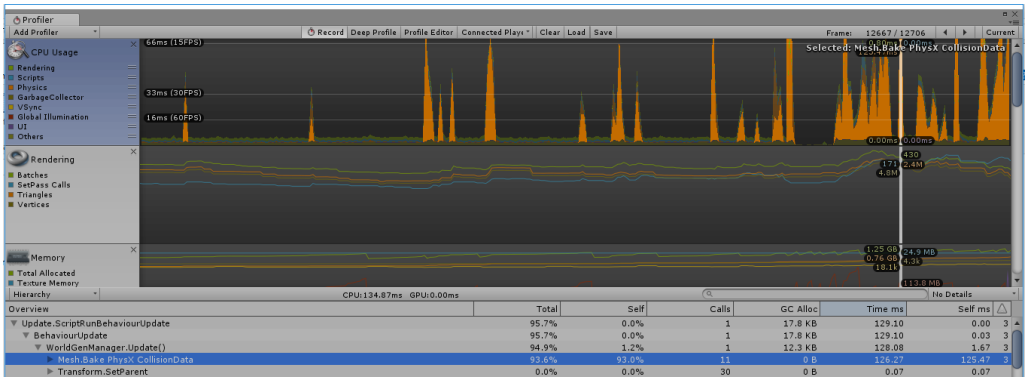


Figure 40: Unity profiler. The orange peaks shows the CPU time used to initialize mesh colliders. The highlighted blue line shows that the game spent 126ms initializing mesh colliders in the selected frame. The target time per update is 16ms for 60 frames per second.

the chunks that are close to animals. We found that this cut the amount of active mesh colliders for the terrain in half. Halving the number of mesh colliders did not sufficiently fix the problem though, so we had to find a way to eliminate them all together.

7.5.2 Local Box Colliders

Our first attempt at not using mesh colliders was based on an approach given by a user from the Unity forums called camander321 [35]. It was based on the idea of placing box colliders into the world on top of voxels. We would only do this for the local space around an animal. The idea is that box colliders are a lot cheaper then mesh colliders, so doing this should improve performance. One issue we had with this approach was the high cost of scanning the local space around an animal for placing box colliders into the area surrounding the animal. If we want to place boxes into a 10x10x10 voxel area surrounding an animal it would become 1000 operations. Scaling with a time complexity of $O(n^3)$ where n is the scan area size in one dimension. We could try to remedy this by only scanning the difference between the current update and the last update. Doing difference scanning of the area would increase the code complexity and still not fix one last problem we have with this approach. The last issue we have is the need for long distance ray casting, the game uses this for various things such as spawning an animal by ray casting at the ground to figure out where the animal should spawn.

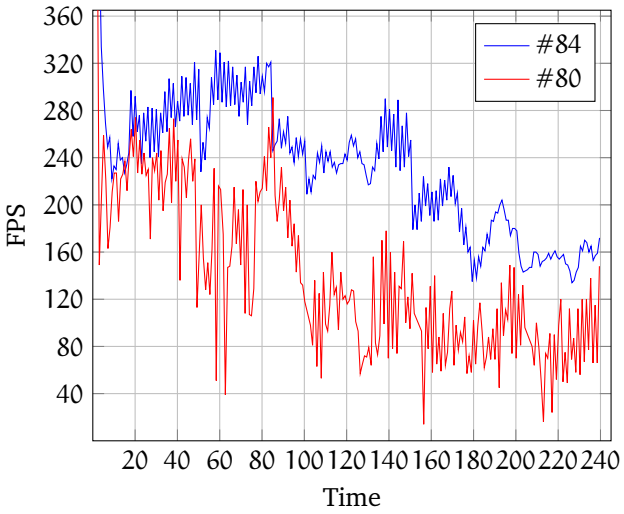
7.5.3 Voxel Colliders

What ended up being our solution to the problem was abandoning Unity's collision system altogether. We would instead rely on the large amount of data we have about the terrain to do our own custom physics. This resulted in the voxel physics system that you can read more about in Chapter 6.5. It is based on using the chunks `BlockDataMap` to do world position to block resolution. This means that for a given world position we can find the block occupying that world position. With the ability to index space we can determine if an animal is colliding by checking the block at the animal position. If the block at the animal position is solid the animal is colliding with the terrain. This is an improvement over the old systems in two ways, it requires no additional initialization work and it does constant time complexity collision detection between an animal and the terrain, regard-

less of of many blocks/chunks there are in the world. The world indexing capability is also used to do ray casting against the terrain.

One weakness of our voxel physics system is that it can not do collision detection for trees. This is because trees have their own `BlockDataMap` separate from the chunks. We still use mesh colliders for the trees, this is not an issue for frame-rate however as trees have far fewer vertices then chunks of terrain. We could try to solve this by having the custom physics also check the `BlockDataMaps` of trees. Another fix would be making the trees share a `BlockDataMap` with the chunk it is placed on. The latter solution is not favourable because it would cause issues for trees that span across more then one chunk which can happen for trees placed close to a chunk border.

Figure 41: Voxel Physics RealBench Comparison. (Red=Before, Blue=After)



7.5.4 Performance Impact

The voxel physics system fixed all stuttering issues caused by the initialization of mesh colliders for terrain. The red line in Figure 41 shows the frame-rate before this optimization. As you can see there is a lot of spiking which is heavily reduced in the blue line showing performance for the voxel physics update. It also improved the overall frame rate by doing constant time complexity collision detection for animal/terrain collisions as also seen in the figure. The average frame-rate went from 135 to 233 which is an increase of 72%.

8 Usability Testing and User Feedback

We had done a lot of testing ourselves for the project, such as bug testing (see Chapter 5.2.3) and performance testing (see Chapter 5.2.2). What we can not test on our own however is usability. This is because we know how the game works as a consequence of having developed it, we need someone without a connection to the project to get unbiased feedback on the usability. For this reason we decided to create a playtest build along with a questionnaire at the end of the project. This happened at a stage in the project where development was already finished, the feedback is used as material for future work for the game. We address only the criticism from the questionnaire here. To read the complete feedback from the playtesters see Appendix D.

Some of the usability concerns we wanted insight for was:

- Usability of the game world.
- Usability of playable animals.
- Usability of user interface.

8.1 Game World

With the game world we wanted to find out if the players managed to beat the game. We asked in the questionnaire if they managed to beat the game and what they thought about the length of the game. We also asked about their opinions on the terrain in general.

Reaching the End

Only one of our 3 testers managed to beat the game, and he spent 49 minutes doing so. These results surprised us, because we can beat the game in 15-20 minutes, which shows the need for external playtesters. They provided feedback saying that they had trouble walking in the right direction, and one playtester thought our wind mechanic was an invisible wall. This shows that we have not sufficiently communicated our game mechanics to the player. The game has some clues to guide the player in the right direction, the wind for instance always blows towards the center. Moving against the wind direction will lead you the correct way. The playtesting showed us that this alone is not enough. The game should also more clearly communicate when wind is stopping your motion, maybe with a special animation for when you are caught in the wind.

Time Spent Reaching the End

They also thought that the game was too long, which we understand from the results of the first question. We intended one play through to last for roughly 15-20 minutes. Adjusting the length of the game can be done through changing one variable, which makes this concern easy to address.

The Terrain

The one criticism we received for the terrain is that the shallow water bodies could be difficult to swim in. To address this we could either make the water bodies deeper, or better equip the water animals to navigate shallow water bodies.

8.2 Animals

With animals we wanted the players opinion on every animal type, and the animal switching mechanic. Below is the criticism we received for each animal:

Animals Types

1. Land Animals: A little slow.
2. Water Animals: Slow on land. Hard to stay inside floating water bodies.
3. Air Animals: A bit slow when flying against wind direction.

As for the fish being slow on land, this is intentional. Maybe we could adjust the speed of the land animal, and the penalty for flying against the wind. There was a consensus among the playtesters that the air animal is the best animal to play as. A view we understand as they can fly, making them the most powerful animal when it comes to achieving the goal of the game. The wind mechanic (see Chapter 6.10.2) is in place to prevent the player from flying all the time for this reason. A general comment about the animals was the desire for an auto run button. Right now you have to hold L-Shift to run, implementing a toggle functionality for running should be easy.

Animal Switching

A criticisms of the animal switching mechanic (see Chapter 6.10.1) is the lack of feedback. To switch animals you have to look at another animal and be within a certain range, then left click. If it fails the game provides no feedback, we should at least give some indication that the player input was registered but the action failed. We could also give some indication about the degree of failure, meaning some indication about how far away you were when trying to switch, and how close you should be.

8.3 User Interface

We asked the playtesters about the UI in general and the settings UI and animal collection UI specifically.

Settings UI

One playtester wanted options for changing the graphics quality from the in game settings menu. We think he was referring to the graphics settings provided by Unity in the game launcher. We could look into adding support for this in the future.

Animal Collection UI

The animal collection UI got 2 request from the playtesters; ability to navigate menu with the arrow keys and a slideshow-slider at the bottom to give more overview of the collection. Implementing arrow key navigation should be simple, the last request however is more involved. We understand the need for more overview, if you collect 30 animals for instance, finding a specific animal can take some time.

9 Deployment

Building a game in Unity is straight forward, we only have to select a platform and architecture to build for, as well as whether it is a development build. A development build allows for us to get warnings and errors that would usually show in the Unity console, shown in a box in an overlay in the game.

We created a build for Windows x86_64. We only created a build for Windows because we did not want to release for a platform which we had not tested the game on, and we could not test on Linux or macOS as neither of us had a machine with these operating systems that also met the requirements for Unity 2017.2.

The game was made available as a release in the GitHub repository for the project (see Appendix A for repository link) as a zip file containing the game files and a readme file. We were originally considering releasing the game on Steam through Steam Direct, but decided against it due to the cost and waiting time before release required by Steam Direct [36].

10 Discussion

10.1 Results

10.1.1 Completion of Initial Plan

The resulting game ended up meeting our expectations to a large degree. We implemented every feature we had planned in the project plan and more (see Appendix B). Our implementation differ from the plan in some ways, animals for instance were planned to only be animated through inverse kinematics. During development we found this to be insufficient because controlling the posture of the limbs with inverse kinematics alone is hard. This prompted us to also implement forward kinematics for the animations, where controlling posture is easy. The wind feature was also not planned initially, we ended up implementing it because we saw the need for it. Audio was originally a stretch goal for the game, we ended up implemented it because the game felt incomplete without it. The game has a complete world rendered with custom shaders, not having audio to complement that gives the game a lacking atmosphere.

The fact that we implemented all of the initial features shows that we could have been a bit more ambitious with the initial plan. Our reasoning for not planning more at the time was that we felt insecure about how much time corruption and animals would take. The corruption feature was meant to be the feature that set our game apart from other games in the genre, by having the world generation change over time. The fact that the feature is uncommon meant that we did not have a lot of resources to draw from. As for the animals, the fact that they had to be generated, animated and functional in the game world is what caused insecurity. We had little experience with animation from before, and the process of bringing a mesh to life in general.

10.1.2 Final Performance of Game

We ended up spending much effort on the performance of the game. This was something we suspected might be the case when planning the game, due to generating the world at runtime. We were concerned with the speed of terrain generation, specifically that it would not be quick enough to keep up with the player. We feel like we managed to bring the performance to an level we are happy with in the end. The game can be played on a modern laptop with a quad-core CPU without the player outrunning the terrain. One of our play testers even played the game with integrated graphics (Intel HD Graphics 630) and reported playing the game with a frame rate of 40, which we feel is good for the hardware. Although the performance is good, the system resource usage is also high. Playing the game on a *AMD Ryzen Threadripper 1950X* will give a CPU usage in the 30-50% range depending on biome. Desert biomes are cheaper resource wise than mountain biomes for instance, due to difference in amount of voxels.

Our system resource usage seems high compared to other games such as Minecraft. Minecraft is known to run on most hardware, and also features a procedurally generated world. Some of the difference in performance is probably due to the difference in devel-

opment time and resources. Minecraft is owned by Microsoft, has been in development since 2009 [37] and is still being updated. Whereas we had 4 months of development and a team of two developers.

10.2 Evolution of Process

10.2.1 The Introduction of Custom Development Tools

We initially had no plan of creating benchmarks or a debug tool. The benchmarks is something that ended up feeling necessary, we needed a way to make sure that our attempts to optimize actually were improvements. The benchmarks gave us good objective measurements for the impact of code changes we made. They also ended up gamifying the development process, the benchmarks scores (time to generate and FPS) became our high scores when optimizing. This made optimizing the code fun, because we would try to see how much performance we could get out of our code and in turn how high of a score we could achieve.

When it comes to the benchmarks we wish we had some better foresight. We ended up making some changes to them over time, this is problematic when it comes to comparing results with older versions of the game. In some cases when the changes were minor in terms of lines of code, we would manually apply them to older versions for comparisons. We should have spent more time thinking about the requirements for the benchmarks when we first implemented them, so that the amount of changes made would be minimal.

The debug tool was introduced as a result of being fed up with writing print statements in the code whenever we encountered a bug. Before the tool we would put print statements in the code we suspected of causing issues. The first approach had the issue of needing to reproduce bugs, because we would not have the print statements ready in advance of encountering a bug. The debug tool can be enabled/disabled at will, and is always implemented making the debug process more convenient. The debug tool is inspired by similar tools found in other games such as Factorio [38], Factorio also has features in-game for exposing the internal game state for the purpose of debugging.

The introduction and use of these tools have given us an appreciation for developing tools used in the development, as opposed to only developing the actual product.

10.2.2 Moving from Google Docs to ShareLaTeX

When we first started writing, we were working in Google Docs. This was fine when the document was fairly small, but as the document grew larger we found that it did not scale very well. While Google Docs is very simple to use, it is not a very powerful tool. It does not support automatic referencing, so we would have to keep tabs on all the references in the document, and change them manually if we added a chapter or a section somewhere in the middle of the document. We had been recommended to use LaTeX to write the thesis. We ended up using ShareLaTeX as our editor as it allowed for real time online cooperation when writing, which was one of the main reasons for us using Google Docs in the first place. Using LaTeX meant that we could have automatic referencing of figures, listings and other chapters so that we didn't have to do this ourselves. There was also an existing template we could use, which made the process of moving to LaTeX quite simple. At the time when we decided to switch over, we had not yet written to much in the thesis, but had already gotten to the point where using Google Docs was becoming difficult.

10.3 C# and Unity

When we were originally deciding what engine we were going to use for the project, one of the reasons for using Unity over Unreal Engine was the fact that Unity uses `C#` , which handles memory management for us, as opposed to C++ where we would have had to do all memory management ourselves. In the end, we still ended up doing a lot of manual work around managing the memory of the game because of the large amounts of data we were processing and storing. An example of this would be the BlockDataMap, which we explain in Chapter 7.2, where we changed the BlockData objects from classes to structs, so that they would be stored sequentially in memory, leading to fewer cache-lines being needed to store all the BlockData for a chunk.

In Unity we also have to manage memory for the Unity `GameObjects` manually. Unity will maintain a reference to every `GameObject` even if our scripts do not. This prevents the `C#` garbage collector from collecting them, which can cause memory leaks. Unity does provide an explicit `Destroy(...)` function for this issue. This makes it so that we manually create and destroy Unity specific objects, just as with normal C++ objects. Normal non-Unity objects in our code however gets garbage collected.

Unity 2017 and earlier versions does not have threading support [28], this means that we could not interact with the Unity API in the worker threads where we were generating our chunks and animals. To work around this, we ended up creating intermediary objects to transfer data from the worker threads to the main thread. For meshes, we had the `MeshData` class, which contained all the data necessary for creating the actual mesh. This data was generated on the worker threads, then sent back to the main thread so that it could generate the actual mesh. Unity also blocks us from using the `System.Collections.Concurrency` library, so we had to create our own thread-safe containers to get around this. While having to do these things to make use of multithreading was a bit annoying, it wasn't that big of an issue and it didn't take us to much time to work around it.

10.4 Future Work

10.4.1 Working with User Feedback

For us, the most major usability concern after doing the usability testing with external testers is the communication of game mechanics. We should make it more apparent from inside the game which direction you should move to progress. The wind should also be communicated to the player more clearly, so that it is obvious when flying against the wind that the wind is stopping you. One way to more clearly communicate the wind, could be to have additional particle effects around the player when they are in the wind.

10.4.2 More Procedural Generation

We originally considered having the game be 100% procedurally generated. The music, as well as the skybox were never procedurally generated, meaning that we didn't quite reach the 100% goal. We knew that procedurally generated music and audio was unlikely to happen, because we felt like that could be enough for an entire thesis by itself. We also ended up using assets for the skybox, this was mostly because we didn't have enough time to generate it procedurally. Making the audio and the skybox procedurally is definitely a thing that could be done in the future.

10.4.3 Better Victory Event

When reaching the end of the world the game displays the UI with a message saying you have won. This is not particularly exciting. As future work we would like to make a bigger deal out of winning instead of just telling the player that they have won. Some special effects should be used to make winning feel like a more special event.

10.4.4 Code rewrite

Unity 2018: Job System and Entity Component System

On May 2nd 2018, Unity 2018 was released, introducing the Job System, a multithreading system for Unity. This system should be very well performing, and potentially better than the multithreading system we implemented ourselves. Unity 2018 also introduces the ECS (Entity Component System), a data-driven design which according the Unity Team is much more efficient than the current object oriented design. We believe that this system could lead to significant performance improvements to our game, so exploring this further, and potentially implementing it would definitively be something we would do. Moving to ECS and Jobs would be a fairly hefty change, which would require rewriting most of the codebase.

Refactoring

There is some code refactoring work left to be done in the project. This became apparent to us as we were writing the thesis and going back to older code. The code surrounding `AnimalState` for instance is the most pressing in this regard. The animal state class was introduced as a temporary solution while working on the separation of the `AnimalBrain` and the `Animal` class. We forgot to fix the temporary solution before moving on, and after some time we had built additional functionality on top of it. This made it so that we could not focus on fixing the `AnimalState` implementation within the time frame of this project. We would like to fix it in the future however, if the project is developed further.

11 Conclusion

We had a goal of creating a game with a procedurally generated world and animals. We feel like we have achieved this goal, our game features a terrain with multiple biomes and animals. We wanted to learn more about noise functions, which we have done through studying and implementing them. The fact that we use more then one noise function also helped us gain insight into the field of generating noise in general.

The process of doing this project has also made us grow as developers. We have never approached a project with a similar degree of professionalism before. We started documenting our meetings and development from day one. We also started logging our time after the first meeting with Mariusz. This resulted in us having a large amount of data that we could base this thesis on, and allowed us to write about the evolution of the implementation.

We had a goal of releasing the game in some capacity at the end of the project. The game would be released as a cheap paid download or for free depending on the outcome. In the end we did not want to charge money for the game, because we feel like the gameplay does not justify it in its current state. The game was released for free on GitHub instead, and we also open sourced the project.

Making *The Grass is Always Greener* has provided us with a lot of experience with larger projects and procedural generation, and it has taught us a lot about writing more efficient `C#` code.

Bibliography

[1] Fugl. https://store.steampowered.com/app/643810/Fugl/. (Visited 2018-05-11).

[2] Superflight. https://store.steampowered.com/app/732430/Superflight/. (Visited 2018-05-11).

[3] Definitition of "biome". https://www.merriam-webster.com/dictionary/biome. (Visited 2018-05-06).

[4] 2017. System requirements for unity 2017.2. https://web.archive.org/web/20171017210246/https://unity3d.com/unity/system-requirements. (Visited 2018-04-30).

[5] Gameobject. https://docs.unity3d.com/2017.2/Documentation/ScriptReference/GameObject.html. (Visited 2018-05-09).

[6] Monobehaviour. https://docs.unity3d.com/2017.2/Documentation/ScriptReference/MonoBehaviour.html. (Visited 2018-05-09).

[7] Knight, N. 2016. Does c# support multiple inheritance? https://stackoverflow.com/questions/2456154/does-c-sharp-support-multiple-inheritance. (Visited 2018-05-08).

[8] Documenting your code with xml comments. https://docs.microsoft.com/en-us/dotnet/csharp/codedoc. (Visited 2018-05-13).

[9] 2018. voxel. https://www.merriam-webster.com/dictionary/voxel. (Visited 2018-04-15).

[10] Bergensten, J. 2011. Minecraft from a developers perspective. https://www.youtube.com/watch?v=dTFkmfnkCfk&t=20m50s. (Visited 2018-04-15).

[11] 2009. Height map definition. http://pcg.wikidot.com/pcg-algorithm:heightmap. (Visited 2018-04-15).

[12] 2018. Relations, functions, and function notation. http://www.ltcconline.net/greenl/courses/152a/functgraph/relfun.htm. (Visited 2018-04-15).

[13] Ares Lagae, Sylvain Lefebvre, R. C. T. D. G. D. D. S. E. J. L. K. P. M. Z. 2010. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8), 2579–2600. doi:10.1111/j.1467-8659.2010.01827.x.

[14] Perlin, K. 1985. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), 287–296. doi:10.1145/325165.325247.

[15] Perlin, K. 2002. Noise hardware (siggraph 2002 course 36 notes, real-time shading languages). https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf. (Visited 2018-04-24).

[16] Gustavson, S. 2005. Simplex noise demystified. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. (Visited 2018-04-25).

[17] Flick, J. Simplex noise, keeping it simple. http://catlikecoding.com/unity/tutorials/simplex-noise/. (Visited 2018-01-12).

[18] 2014. Perlin noise - procedural shader. https://forum.unity.com/threads/perlin-noise-procedural-shader.33725/#post-1510642. (Visited 2018-05-10).

[19] Quilez, I. 2013. Clouds. https://www.shadertoy.com/view/XslGRr. (Visited 2018-05-10).

[20] Bridson, R. 2007. Fast poisson disk sampling in arbitrary dimensions. *ACM SIGGRAPH'07 sketches*, 1. doi:10.1145/1278780.1278807.

[21] 2015. Why do t-junctions in meshes result in cracks. https://computergraphics.stackexchange.com/a/1464. (Visited 2018-04-24).

[22] Lysenko, M. 2012. Meshing in a minecraft game. https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. (Visited 2018-04-03).

[23] Sunday, D. 2012. Lines and distance of a point to a line. http://geomalgorithms.com/a02-_lines.html. (Visited 2018-01-29).

[24] Prusinkiewicz, P. & Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

[25] Compton, K. 2017. Practical procedural generation for everyone. https://www.youtube.com/watch?v=WumyfLEa6bU. (Visited 2018-01-20).

[26] 2018. Transform. https://docs.unity3d.com/ScriptReference/Transform.html. (Visited 2018-04-21).

[27] Thorne, C. Nov 2005. Using a floating origin to improve fidelity and performance of large, distributed virtual worlds. In *2005 International Conference on Cyberworlds (CW'05)*, 8 pp.–270. doi:10.1109/CW.2005.94.

[28] Multi-threaded usage of unity api. https://forum.unity.com/threads/multi-threaded-usage-of-unity-api.348072/#post-2253580. (Visited 2018-05-14).

[29] Physics.raycast. https://docs.unity3d.com/ScriptReference/Physics.Raycast.html. (Visited 2018-05-06).

[30] Rosen, D. 2014. Animation bootcamp: An indie approach to procedural animation. https://www.youtube.com/watch?v=LNidsMesxSE. (Visited 2018-01-22).

[31] Juckett, R. 2009. Cyclic coordinate descent in 2d. http://www.ryanjuckett.com/programming/cyclic-coordinate-descent-in-2d/. (Visited 2018-05-10).

[32] Microsoft. 2018. Texture filtering with mipmaps. https://msdn.microsoft.com/en-us/library/aa921432.aspx. (Visited 2018-04-23).

[33] Strugar, F. 2009. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of Graphics, GPU, and Game Tools*, 14(4), 57–74. doi:10.1080/2151237X.2009.10129287.

[34] 2013. Avoiding shader conditionals. http://theorangeduck.com/page/avoiding-shader-conditionals. (Visited 2018-05-11).

[35] 2016. 2d procedural voxel collision, how? http://answers.unity.com/answers/1218047/view.html. (Visited 2018-04-10).

[36] Steam direct - joining the steamworks distribution program. https://partner.steamgames.com/steamdirect. (Visited 2018-05-12).

[37] Version history. https://minecraft.gamepedia.com/Version_history. (Visited 2018-05-14).

[38] Debug mode. https://wiki.factorio.com/Debug_mode. (Visited 2018-05-14).

A Source Code and Other Links

Link to GitHub page with game download and source code:
https://github.com/Hifoz/TGAG/wiki

Link to GitHub repository containing benchmark data visualization code:
https://github.com/Muff1nz/TGAG_PythonScripts

Link to Trello Board used during development:
https://trello.com/b/ggek1i3b/bachelor-procgen

Link to demonstration/showcase video of game:
https://youtu.be/jQ98rQoNjHA

B Project Plan

Project Plan

Group: ProcGen

Game: The Grass is Always Greener

Abstract

Our idea is an exploration game with a procedurally generated world, where the world gradually gets more corrupt/weird as you move further out (Stuff like oceans in the sky, so when you look up you'll see a whale swimming by). The goal of the game is reaching the end of the world.

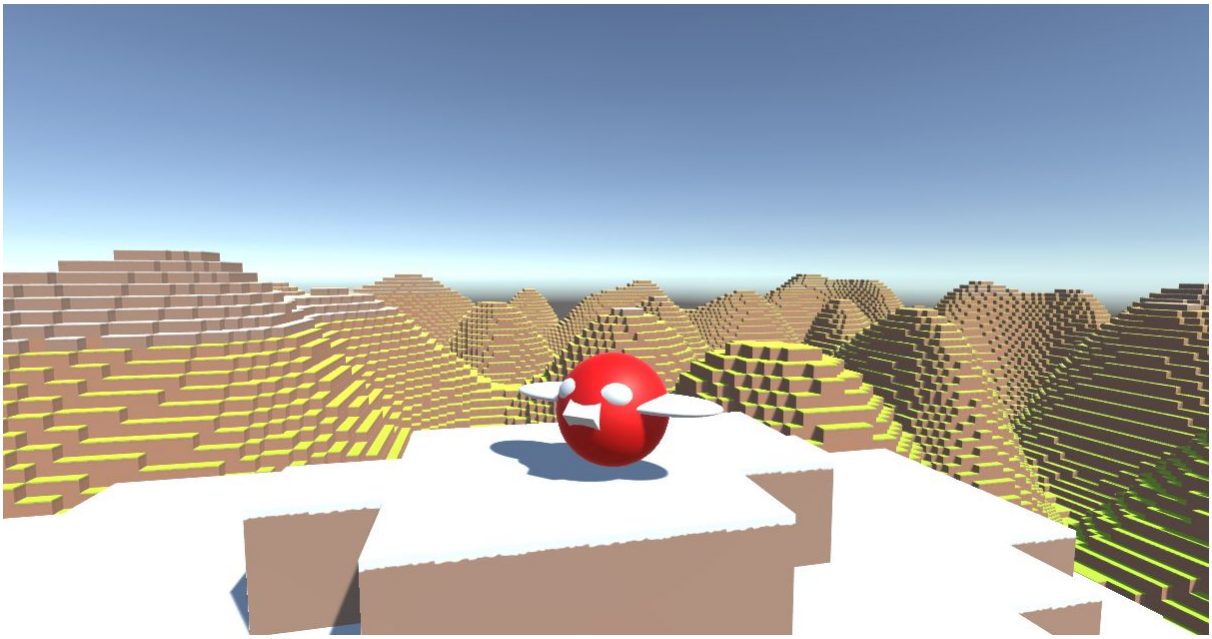


Figure: Picture from an early version of the project.

1. Goals and restrictions

1.1. Background

We came up with the project ourselves because we have a personal interest in procedural generation. We chose this subject matter because we believe that we have a good shot at getting some interesting results, and that we think this might prove an interesting project. The reason for why we think developing this will be fun is because of the iterative process and the emergent characteristics of procedural generation.

Some similar projects/games that inspired us are: Fugl and Superflight, which are games about exploring a procedurally generated world.

1.2. Project goal

The goal of the project is to create a game that revolves around a procedurally generated voxel world. And once we have created the game we wish to publish/release the game in some capacity depending on how proud we are of the end result. If the game becomes good with nice polish (stuff like graphics, audio, gameplay) we might release the game as a paid download on steam for the price of a coffee or something. If the end result is not as good we will release it as a free download.

We also hope to learn a lot about procedural content generation, noise algorithms and inverse kinematics algorithms. The motivation for choosing this project is to get experience from bigger projects. The subject matter it self is another motivating factor as it lets us create a lot of content with few people.

1.3. Restrictions

Our main restriction is time, because we only have 4 months to develop the game and our thesis, which seems really short for normal game development. Most games we hear about take at least 2 years to develop, and with larger teams then what we have.

We are also restricted on our access to assets such as 3D models and audio/sound effects for our game. Neither of us have much experience creating such assets, so we will have to either get them online or try generating them ourselves programmatically.

2. Scope

2.1. Subject area

Here are the main areas that our project will focus on: procedural generation, animation, multithreading, graphics.

For procedural generation there are 2 main activities/algorithms as we see it, runtime mesh generation and noise functions which we will use to do 3 types of procedural generation, parametric, interpretive and L-system generation.

Interpretive generation is what we will use for our world generation, when you interpret some simple dataset, and create something from it, that is known as interpretive generation. We will generate the data from our noise functions and interpret it as environmental data to create meshes.

Parametric generation is when you define some parameters for some entity that is then used create that entity. We want to use this for the generation of animals, so the parameters could be things such as number of legs/eyes or some other feature.

L-Systems is a way to generate strings of symbols that can be used to generate plants using a set of rules for how to grow the string based on its content. We will use this to generate trees and other types of foliage.

Because we intend to generate animals, we need some way of animating them. We will be using inverse kinematic algorithms (IK) to solve this problem, specifically coordinate cyclical descent (CCD).

Due to the computational cost of generating the terrain at runtime, we will be needing to multithread the process to give the user a smooth gameplay experience. This will be a simple producer consumer relationship, where the main thread consumes chunks of the world produced by the world generation threads.

We will be needing some nice shaders so that the world we generate actually ends up looking nice. This is not a core focus for us but something we think would be a nice addition for the overall impression of our game.

2.2. Delimitations

We will mainly be focusing on the procedural generation aspects of the game, so other aspects will not get much attention, such as AI, audio or physics. The reason for these delimitations is that game development is a very time consuming endeavor, so we have to focus our efforts when our time frame is as short as it is.

2.3. Task description

Our task is to create an exploration game based in a procedurally generated voxel world. The player will be playing as an animal that can move around in our procedural world. The animals themselves are also procedurally generated. The goal of the player is to reach the end of the world. As you move further away the world will get more corrupt, and you'll have to eat other animals to become them and progress.

3. Technology

3.1 Engine

We chose to use an engine because engines can handle a lot of the aspects of a game that we will not be focusing on ourselves, such as physics and rendering. We primarily want to focus on procedural generation.

There are two main engines on the market today, Unreal Engine and Unity. We chose to work with Unity because we have more experience with Unity and Unity has a bigger community which makes finding information much easier then with Unreal Engine.

3.2 Languages

C# is the scripting language in Unity, so the majority of our code will be written in C#. For shader programming in Unity ShaderLab, CG and maybe HLSL will be used. ShaderLab is Unitys own shader programming language, CG is also used for shader programming in Unity.

3.3 Additional tools.

- Paint/Gimp for texture and other asset creation.
- Audacity might be used for creating audio assets.
- Git and GitHub will be used for source control.

3.4 Target Platform

We will only be targeting Windows, Linux and MacOS for our game, so there will be no console or mobile support. We chose to support these platforms because they are the easiest platforms to develop for.

4. Project organisation

4.1. Responsibility and roles

Since we are only two people on this project, we will both have the same responsibility and roles.

Our role is as follows:

- Write and develop the game.
- Document our work and meetings.
- Log time use.
- Review the other developers code through pull requests.
- Show up for meetings (on discord).
- Write and develop the thesis.

4.2. Routines and rules in the group

- We will have a sprint review every wednesday.
- Code reviews are required to add code to the master branch of our code repository.
- Always create a feature branch when developing (don’t commit to master).
- Follow coding conventions (see 6.1.3).

5. Planning, Follow-up and reporting

5.1. Main divisions of project

We will be using an agile development model, with weekly sprints. We use a sprint board with a sprint backlog, in progress, review and done columns. Each sprint ends with a sprint review. Before any code can be merged into the master branch, the other team member has to review the code.

The reason for why we are only calling our development model agile, is because it lacks features of other models such as scrum, it is unnatural for us (a team of 2 developers) to have a project owner and scrum master. We can get by with a more lightweight version of agile development.

5.2. Plan for status meetings decision points for the period.

We will have a status meeting every wednesday (sprint review). We will also meet with our supervisor Mariusz Nowostawski every other thursday for a meeting about our thesis. As far as specific decision points go, regarding whether we have to drop implementing certain features and such we have no set dates.

6. Organising of Quality Assurance

6.1. Documentation, standards use and source code

6.1.1 Code documentation

We will comment every function, so that it is clear from the comment what the functions does. If something in the function body is particularly tricky or unclear comment that as well. Our commenting standard is based on:
C# xml comments.
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/recommended-tags-for-documentation-comments

6.1.2 Documentation of work

We will document our discussions from sprint reviews, and designs discussions/decisions. Documenting our thought/decision process this way is really important as it will be a basis for our thesis. We also document our thought process and decisions in our pull requests on github.

6.1.3 Coding Conventions

- Only use *"this."* in reference of members if it is necessary (eg. function parameter with same name as a member variable).
- Use CapitalCase for class and enum names.
- Use camelCase for function names, unless they are static, then use CapitalCase.
- Do not use any prefixes or postfixes on names.
- Always write public and private on member variables, even if one is implicit.
- Enum members are to be all uppercase with underscores between words.

6.2. Risk Analysis (identify, analyze, measures, follow-up)

Identified Risks:

- High computational cost of world generation. **(Probability: M)**
- Having a hard time multithreading in Unity. **(Probability: L)**
- Github crashing **(Probability: L)**
- Not successfully implementing procedurally generated animals. **(Probability: M)**

Consequences:

- High computational cost of world generation. **(Impact: M)**
    - We may have to scale back the world.
    - We have to work on reducing the time complexity of our world generation algorithms.
    - We may need to increase the system requirements of the game, requiring that our users have high core count CPUs.
- Having a hard time multithreading in Unity **(Impact: H)**
    - Without multithreading, creating an interesting world of any scale in real time would seem un-feasible.
- Github crashing **(Impact: M)**
    - Slowing down ability to cooperate.
- Not successfully implementing procedurally generated animals. **(Impact: M)**
    - This would make our game less interesting.
    - We would have to use existing assets for our animals instead of generating our own.

Risk by Rank:
Risk is a combination for probability and impact.

- High computational cost of world generation. **(Risk: M)**
    - This issue is our main concern, because it affects the core feature of our game. If the world takes a lot of time to generate, or it causes an unpleasant gameplay experience (Low frame rate, freezing etc) then that would really diminish our game. The scalability of our world generation will help us with this risk, as it is easy to scale across any number of threads, because the world is subdivided into separate independent chunks that can be handled on its own.
- Not successfully implementing procedurally generated animals. **(Risk: M)**
    - The reason for why this might happen is because neither of us have much experience generating animals procedurally from before, and we lack experience doing animations with inverse kinematics.
- Having a hard time multithreading in Unity. **(Risk: M)**
    - The reason why this could be a risk, is that Unity does not officially support multithreading from its scripting interface. The Unity API calls are not thread safe, so we have to work around that when doing threading. We have successfully worked around it in previous projects, so we are not too worried.
- Github crashing **(Risk: L)**
    - This is not a very big concern, as GitHub is very unlikely to go down, and if it does go down, it won't stay down for so long that it will cause huge issues for us.

7. Plan for Implementation

We will be doing/implementing these activities/systems/algorithms in roughly this order. We feel like it is hard to estimate how much time a given activity will take so we do not have precise time frames or anything like that here. We instead work with the attitude that we will get as much as possible done, and if we manage to implement everything in the below list we will work out new features/systems to implement with our remaining time.

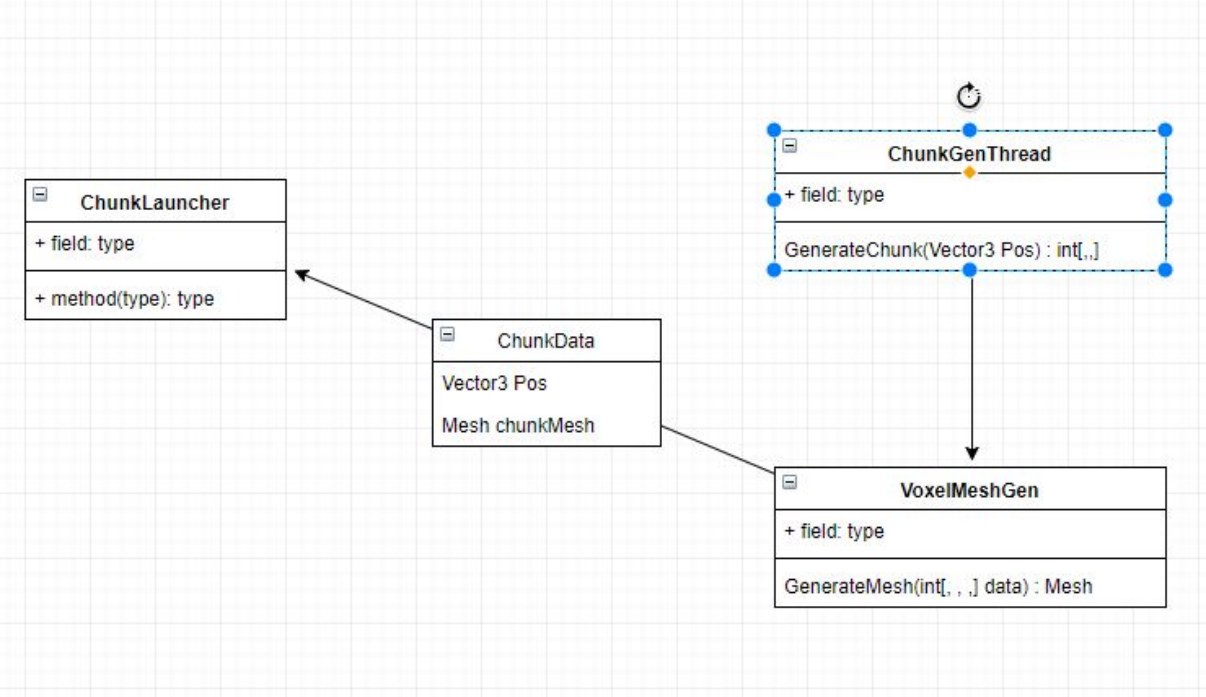
- R&D Prototyping basic procedural generation technology.
- 3D/2D Noise function(s).
- A system for generating chunks of the world in multiple threads.
- An algorithm that can turn a series of cubes into a single optimized mesh.
- A playable character.
- A main menu with an options menu and a play button.
- A system for procedurally generating foliage.
- An algorithm for creating procedural animals (Based on some core features like is it a bird? how many legs? tail?)
- An algorithm for inverse kinematics for procedural animations.
- Some way of getting our desired corruption effect on the world (could be implemented directly in the noise function, or outside).
- Some way of dealing with large worlds and floating point precision.
- Some custom shaders to make the world look somewhat presentable (But this is not a core focus).
- (Stretch goal) Implement audio.

C Meeting Logs

16.01, project plan, core tech for TGAG.

How we deal with chunks and positions: two coordinate systems, the regular worldspace system and a chunk coordinate system. We decide on a size of chunks, let's say chunkSize=10. So we can transform points between these systems by multiplying or dividing by chunksize and flooring. We chose this system because it makes answering questions such as "Is this point inside a certain chunk" easy to answer.

This is an image of our informal UML diagram showing how the core mechanic of generating chunks in threads and launching them into the game world works.



We made a private repository on github for our project, because we might want to release the game at some point.

We decided that we'll start development of TGAG tomorrow (17.01), based on the above informal UML. We will first implement the chunk launcher, using placeholder cubes so that we get the core logic down without the other 2 system in place.

17.01, implementing core tech for TGAG.

We decided not to implement multithreading today, because we want to keep complexity down initially. We decided to make a public static ChunkConfig class, because we want to do real time adjustments for parameters for easy iteration of our world gen algorithm.

We agreed on some coding guidelines: *We will be using camelCase. We will explicitly declare private and public even when not required by the compiler. We will not use “this”, unless required. We will not use “_” for private variables.*

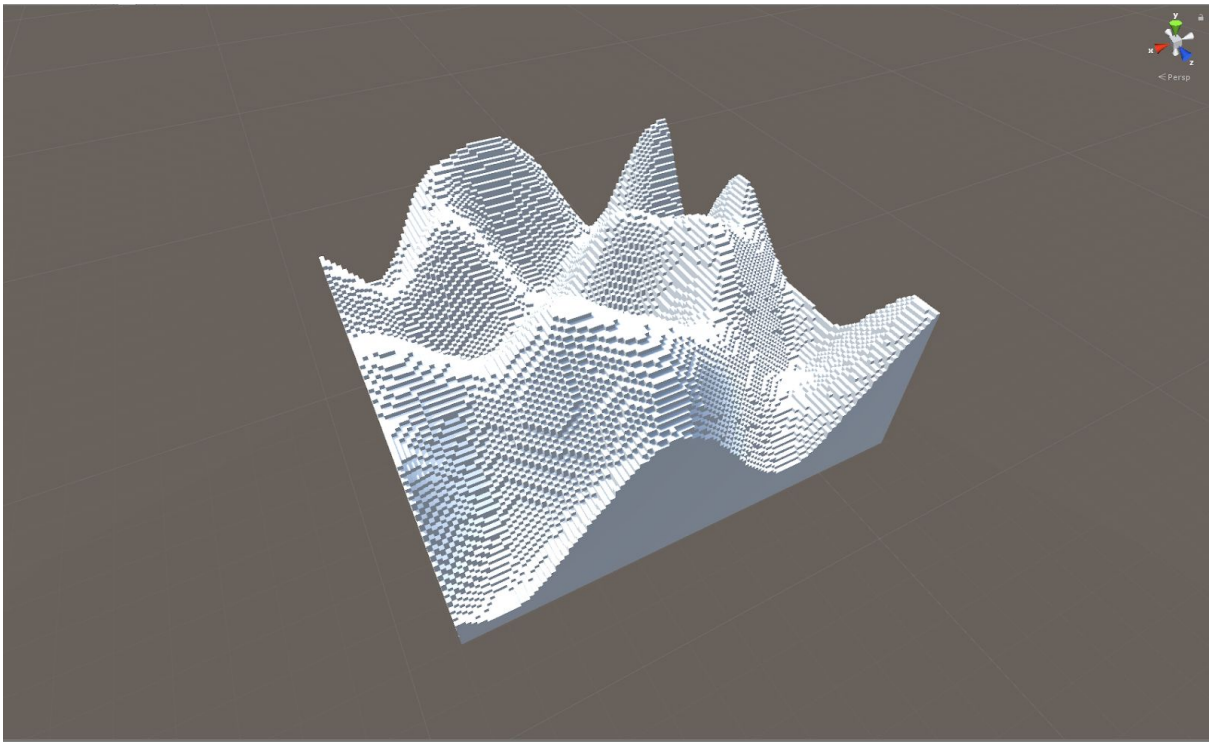
We agreed on an interface for the core tech (Creation and deployment of chunks). Michael is making the chunk voxel mesh class, which takes a 3 dimensional integer array from chunk voxel data that Martin is making. We decided to use a int[,,] to represent the voxel data because we want to support multiple voxel types.

17.01, Sprint review of sprint 1.

We decided to take our interpretation of scrum more seriously at the end of this sprint, we have made some cards for what we intend to do next sprint instead of just winging it.

We we were happy with our use of tools such as discord and screen sharing for collaboration compared to how we have worked at other projects.

We felt like we have made some progress and are happy with what we implemented today (First draft of the core procgen technology.)



The above terrain will generate around the player.

18.01, meeting with Mariusz.

(We should set up toggl or something like it)
We asked about time tracking, he told us that we should track time, and gave us a general impression that professionalism is very important for our thesis. He also talked about the importance of documenting anything, like the meeting logs in this document. Examples of things to document were if a certain problem did not meet our expectations, for instance if implementing something was harder/easier than anticipated. We could also document/talk about what features we spent the most time implementing, how much total time we spent on the project.

We asked about the project plan, it would not be graded and is only there to guide/help ourselves.

Mariusz was positive about the problem area we chose (Procedural generation) because its not only a game, but a technology on its own that we can describe and discuss in our thesis. Talking about stuff like runtime mesh generation and noise algorithms in our thesis gives our thesis more technical depth then just talking about game design for instance (if the game itself was the core focus).

He suggested that we could read Pyroeis which focused on 2D cave generation. They had implemented persistent terrain destruction by recording the changes made during destruction and the seed. He also talked about “moderating” our noise algorithm, which means not just accepting the raw input, but adjusting it based on some rules (for instance if the algorithm creates too high mountains, we could manually shave some height off, or discarding caves that leads nowhere and such).

We asked about the project contract thing, and he thought we would own the rights to the game, so that we could release the game on steam for free/money depending on what we feel like (How good the game is in the end). We could sign the contract and drop by his office to give it to him (he isn’t present on monday/tuesday)

Mariusz asked if we had access to the Unity build tools, and suggested doing weekly builds to show that we can handle a professional workflow.

Note to future self, take notes during the meeting next time maybe.

21.01, Mid sprint meeting.

We decided to remove the chunk caching we used before we implemented multithreading, because the memory footprint would be too big.

We have finished our programming goals for this sprint at this time, and we decided we would spend the rest on the sprint on documenting and writing documents (Project plan, design document).

Math for memory footprint:
Vector3 = 4 floats (x,y,z and kEpsilon(this variable is undocumented?)).
1 float is 4B.
With chunks at 10 in depth and width, and 50 in height there are in a worst case scenario about 40’000 vertices (4*6*25’000cubes).
That gives 40’000*4B=160KB per chunk. After loading 10’000 chunks, there will be 1.6GB for vertices alone, and that is with a chunk height of 50, which we are likely to increase in the future. And this is just for vertices, Adding in texture coordinates and triangles will also increase the footprint.

24.01, Sprint review of sprint 2.

We see that we planned to little work for this week sprint, so we want to plan more work for next week.

We started toggling this week, which showed us how little we work, Martin only worked 9 hours and Michael did better with 16 hours. We hope that planning more work for the next sprint will help us get more hours “toggled”.

Even though we see that we worked to little last week, we did finish all of our planned tasks which is nice.

We discussed the design of water, we will implement it as “blocks” of water, instead of the traditional plane. We need it this way to get our desired corruption effect, with water that does not obey the laws of physics or reason.

We discussed how to deal with foliage, “baking” them into the chunks vs populating the world with foliage in a second pass. We decided to use the second pass because it makes it easier to deal with problems such as “A tree that is in between 2 chunks”.

31.01, Sprint review of sprint 3.

Our planned work amount was better this week, we did not finish our backlog halfway through the sprint like last time. Martin had to “invent” one new card to complete the sprint, and Michael was a bit too ambitious with his planned backlog, on account of procedural textures being more work than anticipated.

We both did better with recorded working hours this week, Michael had logged 21 hours this week, and Martin logged 19 hours, but we still have more room for growth in this area. Next week we will both target working 25 hours each.

We have discussed that we need a better way of handling when to generate what parts of the world. Right now the world is always generating such that the player is at the center, this means that small player movements can trigger a lot of generation. We want to work on implementing a “smart” generation system instead, that has some leeway and the ability to predict and pregenerate sections of the world.

We discussed giving the threads access to the player position, so that they can evaluate if an ordered chunk is still needed by the time they get to the order.

We discussed looking into a custom way of doing collisions, because the runtime generation of colliders for our world is a bit taxing as is.

(Screenshot from sprint 3 version of the game)

01.02, meeting with Mariusz.

Mariusz was not at his office 12:15 :(

07.02, Sprint review of sprint 4.

Last week we set a goal of working 25 hours for this week, we did not reach this goal, Michael worked 18 hours, and Martin worked 14 hours. This was less than we hoped for, and less then we did last week. We will try to reach the goal again for next week.

In terms for planned work amount, Martin did not finish all of his cards in time, but this is more because of not working 25 hours, then planning to much. Michael continued his work on last weeks cards. The done column for this week's sprint is rather unpopulated, but that is not entirely weird, because the features we were working on this week are some of the harder features (procedurally generated animals), which we listed as a risk in our project plan.

In the next sprint we'll work out how many sprints we have in total.

12.02, Mid sprint meeting.

We talked about how we should use our time developing, concerning implementation of new features vs improving existing features. We decided to focus on new features over improving old features, for the sake of risk reduction, we think it's better to find out early if a new feature is hard/time consuming, which we will if we start working on them earlier.

We also decided to get a benchmark scene running, so that we can test if our code changes result in any performance gain. We talked about implementing a different solution to deal with multi threading, ThreadPooling instead of our current purpose built threads. Having a thread pool would make our code more maintainable we think, because then we won't have to "jump" through as many hoops to get a piece of code to run in another thread. Now we have to pass objects back and forth to our threads using queues, which is hard to scale compared to a simple thread pool.

We also talked about starting to write on our thesis, we decided that it's better to write bad material now, then writing no material now, because we can just remove bad stuff in the future.

We talked about our use of static classes such as "public static class ChunkConfig", we know that global variables can be controversial, so we might want to look into an alternative if we feel like it.

13.02, Bachelor Thesis meeting.

We started writing the bachelor thesis, and made a plan to allocate 5 hours to writing the thesis each every week. 4 of these hours would be spent writing independently, and 1 of them would be spent reviewing what we had written together. We will be using trello cards to allocate thesis writing work, like we do for programming.

14.02, Sprint review of sprint 5.

We worked out that we have about 10-11 more sprints of game development time for our thesis, since we want to spend the last couple of weeks working more on the thesis.

We failed to meet the 25 hours of work goal again, Michael worked 19.5 hours and Martin worked 20 hours this week. Although we didn't meet our goal we improved over last weeks result. And when we start using the 5 hours of thesis work we discussed yesterday we will be meeting our goal, if we keep programming as much as we did this week.

What we have planned to do for next week is: BenchmarkScene, Water/River generation, Smart generation, ThreadPooling/Threading refactor and continued work on procgen animals.

(Screenshot from sprint 5 version of the game)

21.02, Sprint review of sprint 6.

We worked on what we said in the last sprint, ThreadPooling was discovered to be a solution inferior to our current implementation.

We are getting closer to our 25 hour goal, Michael worked 21 hours last week and Martin worked 23 hours. We also both wrote in the bachelor thesis this week which is something we said we would last week.

Martin decided to discard support for more then 2 joints in the legs of the animals, to keep the code simpler.

Next week Martin will continue work on the animals, and some misc things like looking into thread scaling. Michael will work more on water generation and general optimizations.

We discussed having labels on our pull request to make the history easier to go through in the future.

27.02, random sprint 6 meeting.

We discussed how we could do biomes, we would need chunkConfig to be a normal class and not static like it is now, and we could have files for the chunkconfig to parse. We could interpolate between the noise values of two chunks to do a transition.

28.02, Sprint review of sprint 7.

This week we managed to make the game scale across 32 threads, by optimizing cache utilization by using structs for blockdata instead of a class, which will place the blockdata sequentially in memory when used in an array. We did some other smaller optimizations as well.

We made the world generate more interesting water structures this week, by playing with the parameters for our world gen algorithm, and the noise calcululating functions themselves.

We also worked more on land animals, giving them better animations, shaders and variety in meshes.

We have had a goal of working 25 hours each week, this week Michael worked 15 hours, and Martin worked 25 hours. We finished all of the work we had planned to do last week, and we added some extra cards during the sprint to keep busy, this means that we underestimated the workload for this sprint, and we have been more ambitious for the next sprint.

We did not do the 5 hours of work each we decided we would for writing the thesis, Martin did 30 minutes of writing, <insert>

Next week Michael will work on biomes, and triangle face merging optimization for chunks. Martin will mainly work on creating bird and fish animals, and some bug fixing.

Sprint 16 should be our last dev sprint, counted 9 more sprints on calendar (until 02.05).

01.03, Second meeting with Mariusz.

Thesis: it’s fine to write mostly about the technical stuff and the programming stuff that we’ve done. also write about process, how we worked, scrum pull request issues. Seeing our game in context to the world and existing work, references (use references in general also i guess).

He liked the fact that we measure things, like performance through the benchmark, which we can graph and talk about, its a professional and objective thing to do, which improves the thesis.

Performance: look into level of detail, generate things at a distance at lower resolution (LOD).

We could write about things in our thesis that we have not implemented, but just discuss the ideas and how they are used in existing games and how we might implement them.

We could playtest to get user data that we can use to backup game design decisions in our thesis.

Animal lifecycle, sleeping, moving around, saves CPU cycles.

07.03, Sprint review of sprint 8.

We have worked on Biomes, Fish/Birds this week as we said last week, none of these features are completed yet however, but we did not expect to finish these features in one week anyways. Some of the things that we did do and merge into Master was bug fixing (Normals for water, mesh related memory leak). We also made a new benchmark, realbench, made to get us numbers closer to what we'd get in actual gameplay. We also made a new repo for Python scripts which we use to visualise our benchmark data.

This has been our best week in terms of hours worked, Michael worked 24 hours and Martin worked 25 hours, so we are practically at our goal. What was not optimal was how much time we spent writing the actual thesis, Martin did not write anything, and Michael did some smaller edits of the document.

Next week we will just continue working on our current tasks, which is biomes and fish/bird animals.

14.03, Sprint review of sprint 9.

This week we finished the first iteration of biomes and AirAnimals, we also did the "lazy chunk" optimization and "greedy mesh" optimization, which waits until the last minute with enabling colliders.

We worked less this sprint the last sprint, Michael worked 17 hours and Martin worked 20 hours, and we spent no hours working on the thesis. The reduced work amount is somewhat tied to increased workload in mobile.

For next week Michael will work on further optimizing the terrain generation, and Martin will implement water animals, and fix tree spawning issues.

We want to start working on the world corruption effect in sprint 11, because that is the last major piece of tech that the game is missing, so we don't want it to be done in the last minute, this would give us an easier time in April, which will be affected by the mobile project and the need for writing the thesis.

21.03, Sprint review of sprint 10.

This week we implemented Water animals and debug tools, and polished/improved the biomes implementation, and improved terrain generation performance.

We worked even less this week then last week, maybe we are getting burned out after working “full time” for two weeks. Martin worked 18 hours and Michael worked 11 hours, so we have some room for improvement again. Martin worked 1 hour on the thesis, which we haven't worked on in some time.

We have brainstormed some ideas in this sprint review, audio, weather, animal mechanics.

Next week we will work on improving the UI, squashing some bugs, giving some classes names that are more appropriate for their current function, implementing a solution for large worlds and floating point numbers and starting work on the world corruption effect.

28.03, Sprint review of sprint 11.

This week we have worked on some random stuff (Bugfixing, refactoring) and UI.

We didn't work much this sprint either, partially due to easter, Michael worked 13 hours and Martin worked 20. We also worked some more on the actual thesis, and sent an email to Mariusz for feedback.

We feel like we need to focus more on the thesis in the coming month, April will be the last full month before we need to finish.

Next week Martin will work on the world corruption effect, and the thesis. Michael will work on game mechanics revolving the animals, such as mechanics to incentivize the players to change animals, and animal collecting, he will also work on the thesis.

04.04, Sprint review of sprint 12.

We worked on gathering a collection of all the animals you've changed into in a playthrough, so that the player can browse and look at all of the animals during gameplay. We worked on game mechanics incentivizing the player to switch animals as well. We also worked on the world corruption effect, lazy chunk launching and water graphics.

This week we both worked 18 hours, which is not too bad considering the high workload in mobile, Michael wrote about the core procedural generation technology in the thesis, Martin did not write in the Thesis this week. We will have to step up our thesis writing game going forward, and we want to look into LaTeX.

We decided to cut some features from our Trello Todo list this meeting, such as the ability for animals to pick eachother up, weather effects. We cut these things because we need time for writing the thesis, and we consider them to be not important.

Next week we will be writing more in our thesis, and we'll optimize shaders, and do more effects for water. We'll work on audio for our game and we'll finish game mechanics that make the player switch animals. We'll also add a win condition, which is triggered when the player reaches the end of the world.

05.04, Third meeting with Mariusz.

We talked about the thesis in general, here's the feedback (Please add more if i forgot some of it):

In general he told us to bother him about reading our thesis again, after we've added more stuff, then he'd give us more feedback. We could message him on discord or send an email.

Start the implementation chapter with some high level descriptions of our implementation, stuff like our components, classes (Could be WorldGenManager, ChunkVoxelDataThreads, the main high level stuff that defines our implementation.) Libraries, technology we use in general. (Kinofog, we use shaders, utils folder probably contains some stuff that we can mention here).

We could write an entire chapter or subchapter about the WorldGenManager alone, this probably applies to other central classes.

We could write a stand alone chapter about optimizations (Greedy mesh gen would go here in that case i guess. *(edit michael: don't think so, as it is its own "feature"/implementation. I think hemeant for us to write about the final implementation main part, and the process of optimization in the optimization chapter, ofc the greedy mesh generation could potentially be mentioned in there as well, but i think it should also be in the main part)*).

We could write about our debug tool as a professional tool that we made and use.

We could write something at the end of our implementation, metadata stuff about our implementation (Lines of codes, hours worked and more).

Change the chapters in our code to be more chronological, split Requirments into techincal requirements and game design, put game design first. Because our project would start with specifying how the game should be, then we work out the technical requirements for the game. (Technical requirements to me is stuff like, the game should not crash, system requirements, but i feel like it covers more than that, i just can't think of what).

REFERENCES, we should find more, and reference them, 20+ references is a number he mentioned, 10 references is bad, should also not go too overboard on the references.

Formatting, add labes/tags to figures and code snippets, keep the font of the code snippets the same or smaller than the general text. Our image/code snippet to text ratio was alright i think. LaTeX supports referencing figure tags in a nice way.

This isn't something he mentioned, but maybe we should have a definitions/acronym list in the appendices or something like that, where we define terms we use (like hlsl = high level shader language). *(edit michael: could this potentially be placed at the end of the introduction chapter?)*

11.04, Sprint review of sprint 13.

This week we worked on implementing incentives for the player to change animals, like wind and big oceans. We optimized the game graphics and physics. We wrote about the core generation technology in our thesis. We also moved the thesis from google docs to sharelatex.

This week Michael worked 23 hours and Martin worked 31, so we are doing pretty good in this regard for once.

Next week we will make the game feature complete, the missing features are Audio and a win condition, going forward after that the only coding we will do will be bug fixing and polish. We will also be developing the implementation section of our thesis further.

18.04, Sprint review of sprint 14.

This week we implemented audio for the game, made a win condition and other bug fixes and polishing. We also wrote in the thesis, about generation.

Michael worked 26 hours this week, and Martin worked 22, which is pretty good. We are pretty much done coding the game at this point and will only be writing in the thesis from now on.

Next week we have a target of completing chapter 6 in our thesis (at least a first draft), which is the biggest chapter. We made cards for every sub chapter of chapter 6 in our trello board.

25.04, Sprint review of sprint 15.

This is the first week where we have almost not done any programming at all. We've almost exclusively written in the thesis. We made a lot of progress writing chapter 6 (implementation) and our thesis is now at 54 pages.

Michael worked 27 hours this week and Martin worked 32 so this is the first week we've both exceeded the planned working hours. Our previous goal of shifting focus over to the thesis has been successful.

Next week we will finish chapter 6 and start writing the optimization chapter.

02.05, Sprint review of sprint 16.

We only wrote in the thesis this week like the previous week. We ran benchmarks and started writing the optimization chapter, requirements chapter and some more work on the

implementation chapter. We made some small polish changes to the game. We made a playtest build of the game with a questionnaire for playtesting. We also made some changes to the thesis based on feedback from Mariusz.

This week Michael worked 12 hours and Martin 26. The difference is mainly because Martin still has chapters to write in implementation. We feel like we are making good progress with the thesis.

Next week we will try to finish chapter 6(implementation) and 7(optimization), we will also start working on the remaining chapters that have not been started yet.

09.05, Sprint review of sprint 17.

Another week of thesis writing. We got less writing done this week then the last week because of the mobile project which was due this week. We worked more on chapter 6 and the game design chapter.

Michael worked 10 hours this week and Martin 16. This is less than usual and caused by the mobile project.

Next week we will finish everything.

D Questionnaire Answers

TGAG Questionnaire

World

Did you reach the end of the world? If yes, how long did it take? *

- (●) I didn't reach it
- ( ) Andre: .....

What did you think about the length of the game? *

- ( ) To short
- ( ) Good Length
- ( ) Too long
- (●) Andre:
  Was never able to reach the end, and i did not spot a lot of corruption, i tried to follow if i saw something unnatural, but i was stopped by "invisible walls" a lot. this was when i was flying. Was able to pas them somtimes, so i never knew if i hit the world end or if i was just hitting something invisible

What biomes did you encounter? *

- [x] Desert
- [x] Ocean
- [x] Mountains
- [x] Forest
- [x] Hills

Is there anything you wish to share in regards to the biomes? The Grass is Always Greener

A lot of cool shapes=)

Please share your thoughts on the terrain you encountered *

Looks good=)

Animals

Which animal types did you try? *

- [x] Land Animal
- [x] Air Animal
- [x] Water Animal

What did you think about the Land Animals?

Cool but not mobile as the Air animals.

What did you think about the Water Animals?

Slow on land, but nice in water

What did you think about the Air Animals?

Played most as Air animals. nice controls

Did you ever switch animals? * The Grass is Always Greener

- (●) Yes
- ( ) No

Anything else you would like to add about the animals?

Animal Switching (a)

Did you enjoy the switching mechanic? *

- (●) Yes
- ( ) No
- ( ) Never switched

Do you have anything to add regarding animal switching?

Had a tendency to switch to air animal rightaway and stick with it, but it is cool that you can choose. worked well when i wanted to switch.

Animal Switching (b)

Did you know about the switching mechanic? The Grass is Always Greener

- ○ Yes
- ○ No
- ○ Yes, but I did not know how to use it

User Interface

Did you find the UI intuitive to navigate? *

- ◉ Yes
- ○ No
- ○ Andre: _____

Do you have anything to share regarding the Settings UI?

Worked nicely

Did you ever look at the Animal Collection while playing? *

- ◉ Yes
- ○ No

Do you have anything to share regarding the Animal Collection UI?

Worked nicely

Do you have anything else to share in regards to the UI?

The Grass is Always Greener

Worked nicely

Computer specs

Because of the resources needed for procedurally generating most things in the game, we know that TGAG is very resource expensive. So in case there is a correlation between performance and enjoyability, we would like to know the computer specifications used when testing.

What operating system you test the game on?

Windows 10 Pro

What processor did your testing machine have?

Intel Core i7 - 4790K

What graphics card did your testing machine have?

NVIDIA GeForce GTX 970

How much RAM did your testing machine have?

16gb

The last page

Is there anything else you would like to add?

Chill music

Thank you for testing our game and filling in the questionnaire :)

TGAG Questionnaire

World

Did you reach the end of the world? If yes, how long did it take? *

- ○ I didn't reach it
- ◉ Andre: 49,62 minutes.

What did you think about the length of the game? *

- ○ To short
- ○ Good Length
- ◉ Too long
- ○ Andre: _____

What biomes did you encounter? *

- ☑ Desert
- ☑ Ocean
- ☑ Mountains
- ☑ Forest
- ☑ Hills

Is there anything you wish to share in regards to the biomes? The Grass is Always Greener

Please share your thoughts on the terrain you encountered *

Sometimes i found it easy to get stuck in the bottom of the ocean. On the other hand i liked the climbing skills of the land animal

Animals

Which animal types did you try? *

- [x] Land Animal
- [x] Air Animal
- [x] Water Animal

What did you think about the Land Animals?

Good climbing skills, and fast.

What did you think about the Water Animals?

They were very fast

What did you think about the Air Animals?

A bit slow going towards the wind direction.

Did you ever switch animals? * The Grass is Always Greener

- (●) Yes
- ( ) No

Anything else you would like to add about the animals?

Animal Switching (a)

Did you enjoy the switching mechanic? *

- (●) Yes
- ( ) No
- ( ) Never switched

Do you have anything to add regarding animal switching?

Animal Switching (b)

Did you know about the switching mechanic? The Grass is Always Greener

- ○ Yes
- ○ No
- ○ Yes, but I did not know how to use it

User Interface

Did you find the UI intuitive to navigate? *

- ◉ Yes
- ○ No
- ○ Andre: _____

Do you have anything to share regarding the Settings UI?

_____

Did you ever look at the Animal Collection while playing? *

- ◉ Yes
- ○ No

Do you have anything to share regarding the Animal Collection UI?

Maybe it would be easier being able to navigate using arrow keys aswell? I would also like names for the different animals.

Do you have anything else to share in regards to the UI?

The Grass is Always Greener

Computer specs

Because of the resources needed for procedurally generating most things in the game, we know that TGAG is very resource expensive. So in case there is a correlation between performance and enjoyability, we would like to know the computer specifications used when testing.

What operating system you test the game on?

Windows 10 Home

What processor did your testing machine have?

Intel I7-7700HQ 2,8 GHz

What graphics card did your testing machine have?

NVIDIA GEFORCE GTX 1060

How much RAM did your testing machine have?

16 GB

The last page

Is there anything else you would like to add?

Thank you for testing our game and filling in the questionnaire :)

TGAG Questionnaire

World

Did you reach the end of the world? If yes, how long did it take? *

- (●) I didn't reach it
- ( ) Andre: .....

What did you think about the length of the game? *

- ( ) To short
- ( ) Good Length
- ( ) Too long
- (●) Andre: I little to long, but that's because I went the wront direction several times.

What biomes did you encounter? *

- [x] Desert
- [x] Ocean
- [x] Mountains
- [x] Forest
- [x] Hills

Is there anything you wish to share in regards to the biomes?

The Grass is Always Greener

I liked the design/colours. Bright and cheery. Would be cool to experience in VR.

Please share your thoughts on the terrain you encountered *

I like the corrupt/glitchy aspects of them. It's like you landed on an uncharted alien planet.

Animals

Which animal types did you try? *

- [x] Land Animal
- [x] Air Animal
- [x] Water Animal

What did you think about the Land Animals?

A litte to slow, but they traversed the mountains great enough.

What did you think about the Water Animals?

Rarely used them. Kind of difficult to get around when the oceans started to float. There was a danger og falling through the water, and getting caught in slow movements on the dry land below.

What did you think about the Air Animals?

The Grass is Always Greener

My preferred animal, nice to traverse large areas of land.

Did you ever switch animals? *

- (●) Yes
- ( ) No

Anything else you would like to add about the animals?

Would like to have an auto-sprint button, or an option to toggle auto-sprint on. My fingers started to hurt from holding down the shift button for too long.

Animal Switching (a)

Did you enjoy the switching mechanic? *

- (●) Yes
- ( ) No
- ( ) Never switched

Do you have anything to add regarding animal switching?

Maybe som sort of range indicator and crosshair, to help the player know from witch distance they can perform a switch. Would make switching to flying animals easier.

Animal Switching (b)

Did you know about the switching mechanic?

- ○ Yes
- ○ No
- ○ Yes, but I did not know how to use it

User Interface

Did you find the UI intuitive to navigate? *

- ◉ Yes
- ○ No
- ○ Andre: .....

Do you have anything to share regarding the Settings UI?

Would preferr to be able to change the graphic quality from the ingame menu. But I have no idea how diffcult that would be to implement in unity.

Did you ever look at the Animal Collection while playing? *

- ◉ Yes
- ○ No

Do you have anything to share regarding the Animal Collection UI?

The Grass is Always Greener

Maybe an slideshow-slider at the bottom, so the player could easily click on the animal they wanted to view, instead of having to cycle through the whole slider. Would also give an indicator of how many animals there are in the collection. Also an counter that showed how many animals the player have collected.

Do you have anything else to share in regards to the UI?

No

Computer specs

Because of the resources needed for procedurally generating most things in the game, we know that TGAG is very resource expensive. So in case there is a correlation between performance and enjoyability, we would like to know the computer specifications used when testing.

What operating system you test the game on?

Windows 10

What processor did your testing machine have?

Intel Core i7-7820HQ - 2.90GHz, 4 cores

What graphics card did your testing machine have?

Intel HD Graphics 630

How much RAM did your testing machine have?

The Grass is Always Greener

32gb

The last page

Is there anything else you would like to add?

Maybe something that notifies the player when they are heading the wrong direction. In-game alert, or maybe something in the game environment.

Thank you for testing our game and filling in the questionnaire :)

E Benchmark Output

Content of a text file output by `SynBench`:

```
Testing from 1 to 25 threads with a step of 2. (4/26/2018 9:44:59 PM):
Terrain: Enabled
Animals: Enabled
[
Time: 30.19 Seconds | Average fps: 378.22 | Threads: 1
Time: 11.04 Seconds | Average fps: 373.62 | Threads: 3
Time: 6.77 Seconds | Average fps: 383.96 | Threads: 5
Time: 5.31 Seconds | Average fps: 363.45 | Threads: 7
Time: 4.42 Seconds | Average fps: 343.75 | Threads: 9
Time: 3.84 Seconds | Average fps: 338.91 | Threads: 11
Time: 3.40 Seconds | Average fps: 311.94 | Threads: 13
Time: 3.21 Seconds | Average fps: 289.44 | Threads: 15
Time: 3.84 Seconds | Average fps: 165.78 | Threads: 17
Time: 5.30 Seconds | Average fps: 74.30 | Threads: 19
Time: 6.29 Seconds | Average fps: 42.92 | Threads: 21
Time: 6.37 Seconds | Average fps: 35.32 | Threads: 23
Time: 6.63 Seconds | Average fps: 27.00 | Threads: 25
]
```

Content of a text file output by `RealBench`:

```
Testing from 8 to 8 threads with a step of 1. (5/13/2018 7:21:08 AM):
Duration of each run: 30 seconds
Average fps: 281.80 | Generated chunks: 292 | Generated animals: 25
    | Cancelled chunks: 92 | Threads: 8
[
x:1.00|y:596
x:2.00|y:434
x:3.01|y:328
x:4.01|y:313
x:5.01|y:288
x:6.01|y:268
x:7.01|y:241
x:8.02|y:234
x:9.02|y:241
x:10.02|y:244
x:11.02|y:230
x:12.03|y:237
x:13.03|y:229
x:14.03|y:237
x:15.03|y:227
x:16.03|y:292
x:17.03|y:259
x:18.04|y:295
x:19.04|y:272
x:20.04|y:296
x:21.04|y:255
x:22.04|y:302
x:23.04|y:267
x:24.04|y:302
x:25.04|y:269
x:26.05|y:293
x:27.05|y:250
x:28.05|y:269
x:29.05|y:232
]
```