3D Procedural Generation with Adaptive Generation Strategy

Teodor N. Nikolov

Report

Submitted in partial fulfillment of the requirements of the degree of
Bachelor of Science in Computer Science,

May 11, 2018

School of Computer Science and Informatics

Supervisor Frank C Langbein

Moderator Bailin Deng

Abstract

This report will discuss a proposed solution towards implementing a procedurally generated landscape in a video game within Unity 3D. Terrain generation is based on an adaptive approach which takes into consideration the position of a moving observer (the player) in real-time and depending on it determines the region of the landscape that should be generated.

Background about the problem will be provided where the problem is broken down into individual subproblems. Details about what elements are commonly associated with procedural landscape generation as well as relevant work will be given. A specification and design outline will also be presented later, coupled with the implementation details needed for solving the smaller problems that comprise the big problem.

The complete solution manages to solve the addressed problem reasonably well, although there are some issues which are addressed, and possible future work solutions are provided.

Table of Contents

1 Introduction

Over the last few decades, hardware performance has greatly improved, and the gaming industry has made sure to make use of that. New games are constantly being released for people to play and the environments of these games would often push the capabilities of the hardware on modern day systems.

Some years ago, there was an outbreak in indie games where the concept of Procedural Content Generation (PCG) was commonly examined. This method for generating content has been commonly used for shaping the environment in video games and it allows a programmer to design levels without necessarily having the artistic skills required to produce interesting or believable results.

As a game enthusiast, I have played many games that have procedural content generation in them. However, for the most part games implement the method in a static context by running the generation process once during the loading phase of a level. There are few games where the world is infinite, and an adaptive generation technique is used. As such, I wanted to explore the concept further and see what the issues and limitations associated with the technique are.

I have also had great interest in developing indie games. Many ideas come to mind that I envision as visually interesting which I would like to include in a game, one of which is the terrain. However, my artistic skills are not the best and knowing that a procedural generation system can be tuned in a way to produce desirable results, I chose to address the problem.

For this project, PCG will be approached using an adaptive generation method based on a dynamic origin point known as ‘the player’. The idea is to have a constantly generated portion of terrain situated inside an infinite along the X and Z horizontal axes world where generation occurs in real-time and enables the player to traverse the landscape freely. The results of the generation process should be reproducible using a unique value called the ‘world seed’.

2 Background

Games more commonly than ever are beginning to use Procedural Content Generation (PCG) as fundamental engines to populating the game environment with interesting objects and features. The process may vary depending on what the content in question is, but for landscape generation the process relatively straightforward. In this section will be covered the idea of PCG and its application in in video games as a landscape generating system.

Procedural Landscape Generation (PLG) is a subtype of PCG and it revolves around the generation of landscapes. Terrain can be a difficult thing to represent the greater the sought realism. The usage of different primitive elements such as noise, heightmaps and volume are talked about to build the generated terrain and spatial data structures that are used to split up the data are addressed.

The results must be rendered and for this reason mesh construction is talked about in this section and how it is rendered within Unity 3D. The rendering and generation of terrain follows an outward approach discussed in greater detail under adaptive generation.

Please note that further references to PCG will be from the perspective of PLG and using 3D geometric data to represent the landscapes in game levels.

2.1 Procedural Content Generation

Procedural Content Generation (PCG) is a method of creating (*generating*) content in an algorithmic way. This contrasts with the more basic approach, where *manual* human effort is necessary to produce satisfactory results. The word *procedural* refers to the process where a function (or set of functions) is computed, resulting in the final generation of content which can then be included in the domain it is meant for (video game, film, audio, etc.).

The procedural generator implemented in this project is fundamentally based on this approach of PCG. It follows an adaptive generation/rendering strategy, based on the viewing position, implemented for Unity Engine. Its results are also reproducible via a *seed* (the world seed) as input for a seeded pseudo-random number generator (PRNG). Reproducibility was a choice and included it for testing purposes where different implementations and optimization techniques can be tested on the same configuration of the procedural generator. Another reason is increasing the player enjoyment factor by allowing them to reproduce the initial configuration of a world by using its world seed and sharing it. Implementation was also easy.

Commonly, procedural generation revolves around the construction of infinite terrain and as such it is possible to specify in which directions the landscape can be infinite.

2.2 Terrain generation

Terrain elevation is commonly used to represent terrain in a 2D context. Sampled values are generated at different points of a procedural surface using noise generators (a.k.a noise algorithms) and the resulting data is then mapped onto a 2D or 3D grid (heightmap and voxel grid) which represents a region of the procedural surface.

2.2.1 Noise and noise algorithms

In everyday life, noise is deemed to be a nuisance that drowns out other, more meaningful elements one may wish to focus on. However, for certain purposes it can be useful and in the context of video games and film-making, noise serves as the ground on which procedural generation is based.

Noise is, as defined by Ken Perlin in (Perlin, 2001), a mapping of values from R^n to R where the input is an n-dimensional point with real coordinates and the results of this function are values which are seemingly random in appearance. However, noise is not random, but instead it is pseudo-random. The noise function will always produce the same output when given the same input and as such is useful when trying to achieve a random-looking result which one would also like to possibly reproduce.

For noise to be useful in a PCG application, a solution to generate noise is required. Such a solution is called a noise generation algorithm, or noise generator.

> "The basis of almost every procedurally generated landscape is a noise algorithm. Without one, every mountain, valley, rock, and pebble must be crafted by hand. This is certainly possible, but not very feasible for larger maps. Noise creates all of this much faster than a human can dream of, and with greater detail. There are many different algorithms for generating noise…" (Archer, 2011)

Seeing this as being the core aspect of PCG, the source goes on to explain fundamental differences between different noise algorithms. They have been individually tested by the author, giving scores on properties such as Speed, Quality, Memory Efficiency and Implementation Ease. This process was done in the context of generating terrain *heightmaps* (2D grid of values), the concept of which will be discussed later.

Some algorithms suffer in terms of quality. This is mostly because they are non-fractal in nature (i.e. self-similarity does not exist at different zoom levels). Work showing how to more closely approximate realistic terrain using fractional Brownian motion (fBm) in 3D space with Perlin noise and the concept of multifractals is presented in (Ebert, 1994).

Below is a short summary of several of the tested noise generation algorithms compared against one another and a brief justification for the selection of the algorithm of choice for this project (2D Perlin Noise) will be provided. There are other algorithms that exist too with each having multiple possible implementations, but they will not be covered in this report.

Midpoint Displacement

Midpoint Displacement algorithm is natively fractal and is based on a recursive approach of generating data. The algorithm is simple to implement and is valued as the fastest of all five. However, this comes at a cost as memory becomes a constraint during execution as the algorithm requires all data that is generated to be stored in memory for it to be able to compute the next random value. The reason is that each generated value is based on several other, already-generated values.

Diamond-Square

The Diamond-Square algorithm is an improvement over Midpoint Displacement. Regarding performance there is not much difference, both CPU and memory-wise. However, it fixes some artifacts introduced by the earlier algorithm and is therefore capable of producing higher quality results. It is generally easy to implement if familiarity with Midpoint Displacement already exists.

Value Noise

Compared to the prior two algorithms, Value Noise is not natively fractal. It can be implemented to excel at memory performance by being able to produce values on-the-fly and not having to store the result internally. It is slower than Midpoint Displacement but is generally very fast.

Speed and quality are inversely proportional, which means that to gain greater quality, it would have to be at the cost of speed. Internally, the algorithm uses an interpolation function (linear, cosine and cubic) and depending on which is chosen, speed and quality will vary. The algorithm is generally easy to understand and implement and terrain realism can be enhanced by implementing fBm (which is also easy to do).

Perlin Noise

As noted in (Smelik, 2009), Perlin Noise by Ken Perlin (Perlin, Making Noise, 2016) is a noise-generation algorithm, which has become an industry standard as it has become the method of choice when dealing with elevation data. It is typically slower than Value Noise, although if Cubic interpolation is used it is much faster. It performs well for smaller dimensions (1D, 2D, 3D), but further than that leads to considerable slow-down.

Memory-wise, it needs to store a gradient table and a permutation table in memory, but other than that it does not differ much in terms of memory consumption from Value Noise.

The quality of the algorithm is also good and understanding the idea behind it is considered not a troublesome task. However, implementing the algorithm is more difficult than the prior three.

Simplex Noise

Simplex Noise also by Ken Perlin (Perlin, 2001) aims to improve its predecessor, Perlin Noise. Simplex Noise is faster than Perlin Noise which makes it more useful for higher-dimensions, although there is no difference in terms of memory as a gradient and permutation tables are still required.

The quality of this algorithm is deemed the “best”, according to the source (Archer, 2011), although understanding and especially implementing it is considered the most difficult in the comparison.

Algorithm of Choice

In the final implementation of the project, 2D Perlin noise is used with fBm. The first reason is that the noise algorithm provides a very good balance between performance and quality. The algorithm is chosen with multifractals in consideration and as such can be layered using fBm to enable an improved level of control over the landscape elevation, which is included in the final solution.

The second reason for using the algorithm is that it is already implemented inside Unity 3D (Unity3D) which will be later discussed in Rendering and Unity 3D. By being part of Unity’s scripting

environment, it saves me (the programmer) the time and effort required to write an implementation of the algorithm to include in the project. For specific scenarios it may be more appropriate to write such an implementation for potential speed-up, such as in a final product that is sold to the public (like a video game). However, as Unity is essentially a game engine, it is highly optimized to perform real-time work and the implementation of the noise algorithm inside it is efficient enough for the scope of this project.

The third, less significant reason that Perlin Noise is used, is due to it being the considered industry standard. As such, during the initial phases of research, it was the first algorithm I came across and got familiar with. There is a plethora of resources detailing how to generate landscapes (articles, blog posts, papers) and in almost all of them Perlin Noise is mentioned and used. Therefore, I quickly became comfortable with it and in cases I needed to make my own implementation (as opposed to the Unity one), I believed that I would achieve the task with ease.

2.2.2 Heightmap

A heightmap is a 2D grid of values, where each value in the grid is sampled at a specific point in 2D space using its XY coordinates. Often nowadays, terrain generation is based on a fractal noise generator, such as Perlin noise in combination with fbm, first proposed by Benoît Mandelbrot. Fractional Brownian Motion (Ebert, 1994) exists as a solution to more closely represent a realistic, fractal landscape. However, this is costs the CPU more cycles as the data for a single point will need to be generated N-times.

The lack of this influences the elevation of the terrain produced by the project PLG. I did not implement a method for layering differently-scaled noises because I did not have time to investigate that aspect of the initial idea and the produced terrain turned out not as realistic as I had envisioned it.

Heightmaps are commonly visualized by first populating the map with sampled values and then rendering the map to a (usually square) raster image. In the context of PLG, heightmaps often serve as the basis of different models like vegetation, rainfall, temperature and more. However, they are mostly notorious for being used as the initial stage towards generating terrain by shaping the general elevation of the surface. Also layering multiple heightmaps and transforming them into a single result would create terrain with much more complicated features than if only an elevation heightmap is used.

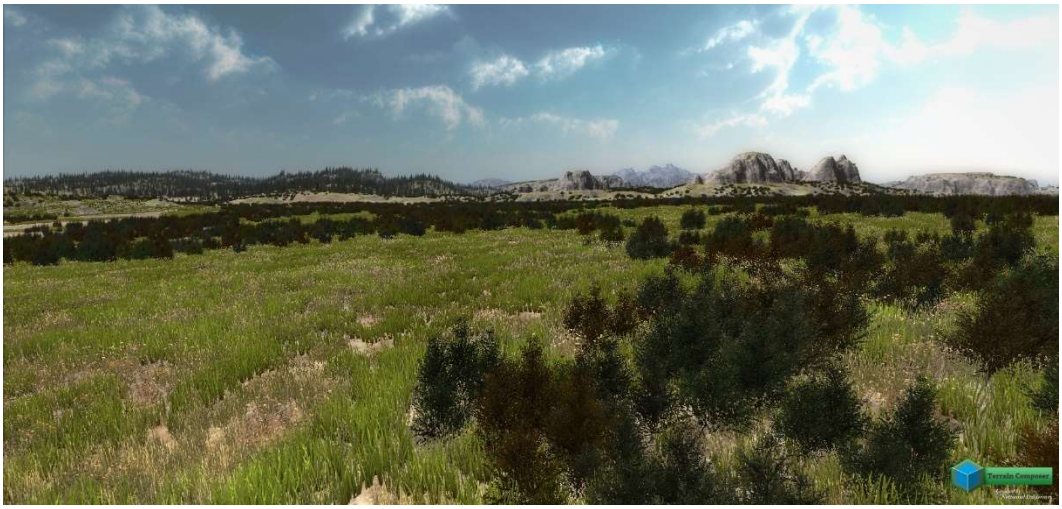


Figure 1 – Example of Procedural Landscape Generation with vegetation, elevation modeling and procedural texturing http://www.terraincomposer.com/procedural-terrain/

In some cases, heightmap values may be treated as a threshold. Commonly with elevation, a sampled value would represent the border at which the surface transitions from *filled* to *unfilled*, or *empty.* Heightmaps are useful in representing simple 3D structures of large amounts of homogenously filled space.

It is good to remember that heightmaps are only an *approximation*. The values on the map are sampled at regular intervals and everything in-between them is unrecorded information.

2.2.3 Voxel volumes

While heightmaps are very useful in PLG, they alone are not enough to represent terrain with more complicated features, such as caves or even floating islands Figure 2. This limitation can be overcome using a voxel-based approach towards generation, as opposed to the 2D heightmap approach. In fact, a game known as Minecraft (Persson) became very popular as it was entirely based on procedural blocks (or voxels).

Figure 2 - A procedurally generated "floating island" in the game Minecraft

A voxel is a point on a regular grid in 3D space which represents a value. This value is comparable to the sample values stored in a heightmap, except that they are 2D. The primary usefulness of voxels is that they excel at representing volumes or spaces which are not be homogenously filled. As such, many are combined to form a grid of voxels that approximates the landscape in 3D.

Voxels do not have an explicit 3D coordinate associated with them, but instead it is inferred from the voxel's relative position inside a volume container which is defined by certain dimensions.

2.3 Spatial data models

Spatial data models are essentially data structures which contain data that defines an object situated in a geometric space. There are many spatial data structures available out there, some of which will now be addressed.

Quadtree

A *quadtree* is a tree data structure based in 2D space. It recursively subdivides space into quadrants and each node of the thee has exactly four child nodes. It is useful for reducing the impact on computational and memory performance. One way it does so is for each cell to represent homogenous space by having many identical samples over a region be treated as one sample. Another way to reduce performance impact is by implementing quadtrees into a level of detail (LOD) system which scales the represented data the further it is located and omits unnecessary details. An *octree* is the 3D space equivalent of quadtrees.

Grid

Grids are another common spatial data structure. A grid is a regular tessellation of 2D or 3D space which divides it into a series of contiguous cells. There are numerous variants proposed to what the shape of a contiguous cell is, but most commonly cells are based on a rectangular shape such as a square or cube.

Data structure of choice

Compared to quadtrees and octrees, grids have one level of detail and as such may not be the best choice if large amounts of data need to be stored. However, the advantage is that implementing the data structure is easier than the rest and this is one of the reasons I chose a grid-based spatial data model.

A second reason is that to quadtrees were a difficult subject to get the grasp on. The time spent to understand how such a data structure works, what its use is and how it can be applied to the context of the project was much larger than I had previously expected.

A third reason why a grid is used is because of the mesh extraction algorithm I have chosen beforehand (discussed shortly afterwards). The algorithm produces meshes based on neighboring voxels and if I had used quadtrees to implement a LOD system to represent the terrain, it would have introduced visible mesh seams at the border between two regions of different resolutions.

2.4 Mesh construction

Generated content needs to be presented somehow. In 2D, heightmaps can be applied to bitmaps to visualize the data. In 3D, a mesh will need to be constructed instead and fed to the GPU to render. A standard approach to doing so is using an iso-surface extraction algorithm. By implementing such an algorithm, one will be able to produce a final mesh that can be rendered by the GPU.

There are several iso-surface extraction algorithms for 3D data such as Dual Contouring of Hermite Data (Ju), Marching Cubes (William E. Lorensen, 1987), Surface Nets (Gibson, 1999) and others. Marching Cubes is the most famous algorithm and has many information available to it. It was also the algorithm of choice for this project because it had a well-known implementation already (LINGRAND, 2002) and I could use it as the basis for my own specific implementation.

"Marching Cubes and other algorithms use a voxel representation of the volume, considering each data point as the vertex of some geometric primitive, such as a cube or tetrahedron. These primitives, or cells, subdivide the volume and provide a useful abstraction for computing isosurfaces." (Philip M. Sutton, 1999)

2.5 Adaptive generation

The infinite nature of voxel grids means that data will also be infinite in size. For use by computers, it needs to be limited somehow to be around an arbitrary viewing position with a certain range. There are two approaches towards managing this process - *static* and *dynamic (or adaptive)* generation.

2.5.1 Static generation

Static generation is an approach used when speed of generation should not matter. If a level needs to be in a confined space (often a rectangular bounding box), all the relevant data can be precomputed and persistently rendered. There is no need to worry about introducing real-time performance penalty, unlike with adaptive generation. The designer is free to decide how many algorithms will be used, how they will be layered and how data is further transformed. The game's "Loading Phase" is expected to take a while.

2.5.2 Dynamic generation

Adaptive generation is the opposite approach of static generation and the idea behind it is that the terrain is generated on-the-fly around a (most likely) moving viewing point. The world is treated as infinite and the region needing to be generated as non-deterministic. It is progressively rendered from the viewer outwards. To enable real-time generation, the designer of the system needs to follow an adaptive generation strategy.

2.5.3 Speed and memory management

Adaptive generation, being the solution to real-time generation, requires that the underlying noise generators are sufficiently optimized for the task and that generated data is properly managed by the system to be rendered.

2.5.4 Outward rendering

For the world manager to have an "outward" rendering style, it needs to utilize a sorted data structure to control the sequence of elements being rendered, according to distance from the generation origin.

A *sorted array* is the simplest data structure to use. It is simple to implement but it is very inefficient because the sorting algorithm may need to iterate over large amounts of data. Performance is generally slow in this context.

A better approach is using a *priority queue* to determine the sequence of elements needing to be rendered. The data structure is based on a self-balancing approach and as such is always sorted. It is faster than sorting an array and implementation is also relatively easy.

Initially I used a sorted array because it was easy to implement, and I could effectively structure the program. I then replaced it with a priority queue to achieve improved real-time performance.

2.6 Rendering and Unity 3D

Unity 3D is the environment in which this project is built. The reason is that the generated data would need to be presented somehow and the choices to for doing so are few. The hardest would be to design and implement a custom rendering engine from scratch. An easier solution would be to use one that is already implemented in a game engine, such as Unity 3D.

Unity is a cross-platform game engine which has as a primary purpose to assist the building of 3D video games or even, albeit more rarely, 2D video games. It has a built-in scripting environment based on C#, but it also supports an external environment, namely Microsoft Visual Studio. I chose to use Visual Studio because the environment offers more tools to ensure the written code is clear and correct, thus saving me time when I need to debug the software solution.

To save time, Unity 3D was the preferred choice for the project as I had already used the program and am comfortable with using C# as a scripting language. In addition, building a rendering engine from scratch, which I have experience with, would have taken a much longer time and it was more important to focus on the other elements of PLG.

3 Specification and Design

The aim of this section is to present the reader with information to better understand how this project has been designed. Methods used for addressing elements of PLG will be discussed here.

When programming the solution, I paid careful attention towards building a well-structured hierarchy of objects and modules. I aimed to keep the code decoupled and to separate my implemented environment from Unity's as much as possible. In a sense, my approach was to treat the written code as a library that could easily be transferred onto another engine and not having to rely on Unity to make it all work.

3.1 Procedural generator

A clear definition of the rules for a procedural generator is not defined. Regarding map (terrain) generation, Amit Patel says that "What we're trying to do with procedural map generation is to generate a set of outputs that have *some* things in common and *some* things different each time." (Red Blob Games, 2013).

For example, a designer of such a system may want it to produce values in a certain range and as such all values will be similar in a way. However, each time the procedural generation is queried with different input parameters, there ought to be no clear correlation between the output results.

For this project, I decided to make the terrain infinite only along the X and Z horizontal axis with the Y axis point up (in the sky). I did not want vertical infinity because the project has been partially inspired by Minecraft where the world height is limited. The generation should be adaptive, so a constantly rendered region should be present around a moving observer (the player). I am also aiming for a flat artistic style.

The terrain uses multiple layers of noise, sampled at different frequencies, to create fractional terrain that is more realistic. Caves, canyons and other more interesting features are not part of the generated terrain because the world manager took more time to implement and I did not have time to properly address terrain realism.

3.2 The player object

A moving viewing point is needed for an adaptive PLG system. Here it is achieved by having a script attached to a game object residing in Unity's scene. The PLG system directly communicates with the script to obtain the position data of the player's location in the world and uses it to calculate the terrain region to generate.

It reflects the interactive element of the environment which interfaces between the user and the internal system. The person playing the project instance can freely move along all three axes in 3D space and viewing is implemented in a first-person shooter (FPS) style. The FPS style limits the rotational capabilities of the viewing point by removing the 'roll' of the object.

The player is associated with having two viewing distances or radii. When extended from the player, one radius (the *near* view distance) is used to form a circular area inside which the terrain is guaranteed to be generated and rendered always. The other radius (the *far* view distance) is used to draw a larger circle around the player and to keep all generated terrain inside its area loaded into memory. This

is useful because if the player's movement is consistently in a small area, it is possible to have a situation where terrain would need to often be generated and discarded. This incurs a performance hit on CPU and memory (via garbage collection) and is better managed using said viewing distances.

3.3 Noise generator class

The noise generator in this project acts as an interface between an algorithm implementation and the object managing and using a noise generation instance. There is currently one such class that wraps Unity's 2D Perlin noise implementation, which takes as input two coordinates. In the context of the program, those are the X and Z coordinates of the viewer's position in world space.

The class provides a finer level of control over terrain elevation using several parameters detailed a bit later. To provide reproducibility, a PRNG is initialized using a seed.

One method that is responsible for taking two coordinate values and outputting a noise value is used, called the Generate method. It includes an implementation of a chosen noise algorithm (in our case 2D Perlin noise called via Unity's scripting API) and it should take the X and Y parameters to be used by the noise generator. It should keep track of the current frequency and amplitude of each octave and apply transformations to the input, output and frequency and amplitude values, all of which will be discussed below.

Generation origin

The noise class uses a 2D vector to represent the generation origin. The origin is used in a transformation operation on the input coordinates for the noise generator and is used to manually select a point of interest from which generation should begin. This was done for debugging purposes and as an enabler for terrain reproducibility which is, by default, randomly generated. It is entirely managed inside the class and cannot be externally modified.

Scale

The scale factor can be specified to govern the resolution or zoom-level of the produced result. It is a transformation applied to the input each time it is specified to the noise generator.

The default is 1, although it is highly desirable that the value is increased to introduce better results. By scaling down the rendered area to cover a smaller region of the landscape to be sampled, more detail is introduced which results in a much more realistic-looking terrain.

Seed

The idea of having a global offset was to be able to specify a point to generate from. A seed value can be specified to the noise generator to be used when initializing a PRNG. By allowing the PRNG to randomly specify the offset of the world along the X axis and the Z axis using a seed, the generated sample for a specific coordinate will always be the same.

Octave

In the previous section it was noted that Perlin noise was used with the consideration of multifractals. To enable the layering capabilities sought, the number of octaves (or layers) must be specified. By default, it is set to 1 layer when instantiating a noise generator, although any number can be

specified. Remember that performance degrades when more samples are needed to compute the final value of a point.

Lacunarity

Lacunarity is a value which is used as a factor to decide by how much the frequency of each consecutive octave should change. It defaults to a value of 2, meaning that for each consecutive octave, the sampling frequency will be twice as high compared to the one for the previous octave.

Persistence

The persistence value goes together with lacunarity. Persistence is the factor by which the amplitude of each consecutive octave is transformed. It defaults to a value of 0.5, so for every octave the amplitude will be twice as small than the previous octave's amplitude.

3.4 Chunks

It is important to specify in which format the world should be for the system to work correctly. In this project, the world in which the player can move is infinite along the X and Z coordinates, but it is limited along the Y coordinate. For simplicity purposes, the world is treated as a 2D grid made of vertical 3D volumes known as *chunks*.

3.4.1 Heightmaps

At the basis of chunk generation are heightmaps. They are used in the program to represent terrain elevation over a square region. Their purpose is only to prepare the data in a 2D format and from there, terrain elevation is transformed into a 3D representation to form chunks which are wrapped in a *chunk object* container.

While I aimed for a truly realistic terrain in terms of caves and similar features, heightmap usage is limited to simple terrain elevation. It is sampled by only one layer of Perlin Noise at a defined scale and offset by a random coordinate. I have not included elements of vegetation, temperature or anything else because development of a memory manager took more time than I expected.

3.4.2 Voxels and Marching cubes

Marching cubes is an algorithm for extracting surfaces which intersect with the edges of individual voxel cells in a 3D world grid. A common convention is needed and is presented to enable the algorithm to correctly extract the surface data contained in a cell.

Voxel

Because Marching cubes extracts mesh data from individual voxels in 3D space, the grid cells of the 3D world must have a value associated with them. This is achieved by considering the 8 corners of each voxel cell to be voxels of Boolean value. That is, each corner of a voxel cell can be "0" when empty and "1" when solid. The other option is for the corner Boolean voxels to represent a density value between 0 and 1 inclusive. The latter is useful when extracting the mesh of a 3D volume because it can more accurately approximate terrain elevation when provided with a threshold value, using interpolation methods.

By chaining the 8 Boolean corner values of a voxel cell in a specific sequence, a byte number is formed which acts as the current configuration for the voxel. This means that there are a total possible of 256 voxel configurations, although it is possible to narrow them down to 14 using inversion and rotational symmetry (LINGRAND, 2002), (William E. Lorensen, 1987). The indexing convention for vertices and edges I followed is the one provided in (Bourke, 1994) and for clarity purposes it is provided here in 13 in relation to the world coordinate system.

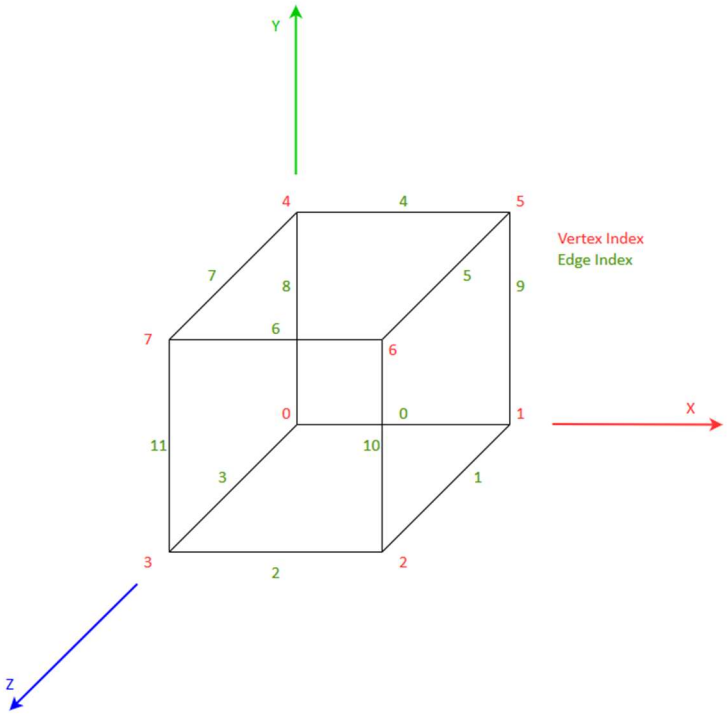


Figure 3 - Diagram of a cell showing its vertex (corner) and edge indices as situated within the 3D world

Marching Cubes

Marching cubes is an algorithm which works on a small grid of neighboring voxels that represents the 8 corners of an abstract cubical cell. The algorithm uses the density values stored in each corner to determine, using a threshold value, which of the corners are inside the surface and which are outside. If there is a transition between solid and empty occurring anywhere within the cell, the algorithm determines along which edges vertices should be placed. This is dependent on the configuration of the cell and the position depends on the implementation of the algorithm. The placement of the vertices is determined by interpolating the corner values. To learn more about how Marching cubes works, please refer to (William E. Lorensen, 1987).

I was going for a specific artistic effect with this project and terrain was meant to follow a flat style. I achieve this by treating voxels to contain Boolean density values as interpolating between 0 and 1 will always result in vertices being centered along the abstract cell's edges. The resulting geometry has polygons whose edges are angled at increments of 45° along the XYZ axis which is what I was going for.

My marching cubes implementation also has internal changes to address the flat style in terms of shading, even though there is a slight cost in memory. Also note that the produced geometry has edge

cases suitably explained in (LINGRAND, 2002). These ambiguous cases I have not had enough time to research because I was more focused on the world manager. I have not included in the marching cubes implementation a way to determine case ambiguity.

3.4.3 Chunk objects

Chunk objects are responsible for generating and storing chunk data, constructing a mesh from the data, rendering the mesh and doing cleanup when it needs to be destroyed. Each chunk object has its own 2D coordinates that represent its location in the world grid, rather than the absolute world coordinates. The chunk object is static and as such its position cannot change.

Chunks are of size 16x16 in the 2D world grid (the reason for which is explained after this subsubsection) and their coordinate values, being unique due to the static nature of the grid cells, are used to compute a hash value which will be discussed later in the world manager.

Figure 4- The chunk object lifecycle

The lifecycle of a chunk object in the context of this project, as shown above, is made of four stages.

Chunk generation

This stage of the lifecycle involves the generation of individual samples of 2D noise and populating them into a heightmap. Then, the elevation data inside is used to populate the 3D grid of voxels inside a chunk object to represent the elevation in 3D.

Mesh construction

To prepare the generated data for visualization, a 3D mesh is constructed. This is done using Marching Cubes which uses the voxel data of each chunk to determine where in the 3D volume a transition from *solid* to *empty* occurs. All geometry-related data is produced during this stage of the lifecycle.

Rendering

Rendering is the process of visualizing the previously constructed mesh data. This is done by Unity and therefore all geometry data needs to be transferred from my environment into Unity's scripting environment.

Chunk removal

This stage is generally the final one for each chunk. During this process, any cleanup that needs to be done is performed and is largely important because it must be done both for my environment and for the Unity environment.

3.4.4 Chunk segment

Because Unity introduces constraints in terms of the geometry data amount per in-game object (65000 vertices max), chunks are broken down into sub-components called *chunk segments* whose internal data format is compatible with Unity. Chunk segments can be associated with only one chunk object and only it can operate on them. They serve no other purpose than to enforce the geometry data limit and to implement communication between Unity and my decoupled environment.

Chunk segments can be vertically stacked to fill a chunk, the number of which is used determines the height of a chunk. By combining the multiple chunk segments the world will visually stretched from the ground to the plane on which the top side of the top chunk segment resides.

To overcome the 65000 vertices per game object limitation enforced by Unity, I decided to make chunk segments be 16x16x16 voxel grids that can be stacked multiple times to allow height to be freely chosen. This makes chunk objects have dimensions of 16xNx16 where N is the number of chunk segments vertically stacked together times 16.

The choice of such dimensions means that one chunk segment will be composed of 4096 separate voxels. Marching cubes must go through each voxel in a chunk segment to produce the triangles respective to it. The maximum possible number of triangles a cell can have is 5 (Bourke, 1994)**Error! Reference source not found.** and by noting that each triangle is made of 3 vertices we get the following calculation for the maximum number of possible vertices per chunk segment:

4096 voxels * 5 maximum triangles * 3 vertices = 61440 vertices total

16x16x16 is the maximum possible size of the grid because if any of the dimensions is set to 17, the resulting number of total possible vertices becomes 65280, which is above the limit and will result in unstable performance, albeit the chance of this occurring is very miniscule.

The chunk segment holds a copy of the generated mesh data as well as a reference to an instance of a chunk instance script (discussed right after this) to enable the piping of mesh data from my internally managed environment to Unity and its game objects. A method for generating a mesh and a method for applying the geometry data to the Unity game object are essential.

3.4.5 Chunk instance

Unity has its own rendering engine which can be used to render the geometry data of the procedural terrain. To interface with its rendering engine, a custom object is used, called a *chunk instance*. Its only purpose is to link the decoupled from Unity chunk segment with Unity itself.

Chunk instances are interfaces between my environment and Unity and act as wrapper scripts for chunk segments. These instances are scripts that are directly linked inside the scene in Unity. Their purpose is to provide a place where the generated mesh data can be transferred for Unity to read and render. Every chunk instance is associated with and represents one chunk segment that is internally managed inside the environment. This is the only way to communicate with Unity – through their scripting API.

3.4.6 Chunk coordinates classes

Chunks and chunk segments on their own do not have coordinates associated with them. One solution was to coordinate values inside the objects as private variables, but for clarity purposes and code structure I decided to have two separate classes that would represent the 2D and 3D coordinates of individual chunks and chunk segments respectively.

Because chunks reside on a 2D grid, their coordinates can be either in world coordinate space (as the one used by Unity), or instead they can be in "chunk" coordinate space. In chunk coordinate space, coordinates are derived from the position of a cell in terms of its row and column indices. In 3D, slices are used to represent the Y chunk coordinate.

The 3D version of the chunk coordinates class is used only by chunk objects and chunk segments for determining the height offset of individual voxels when terrain is being generated for a chunk. The 2D one is used by chunk objects to determine the world position offset of the chunk and to correctly sample noise for each voxel cell that a chunk is comprised of (remind yourself that that voxels do not have a coordinate associated with them directly). 2D coordinates are also used by the world manager to determine the position and distance of chunks in respect to the observer (the generation origin).

Both coordinates classes provide methods for obtaining coordinate values either in world or chunk space depending on which is needed. However, all instances of the chunk coordinates class are formatted in chunk space. This choice was initially somewhat arbitrary, although later I found it easier to work on the system by treating the space occupied by chunks to be in chunk space rather than Unity's world space.

3.5 World manager

With the program being based on an adaptive generation approach, a form of speed and memory management that enables outward generation and rendering of the landscape is required.

A world manager is used to handle individual chunk data by consistently keeping track of chunk objects and managing them. It decides which objects chunks need to be generated, which need to be removed, have their mesh constructed or when they need to be rendered. There were issues where the main thread would get blocked until all chunks have been generated. As such, I decided to thread it using a thread pool.

Outward generation is achieved using a *priority queue*. The priority queue is used to sort the chunks in respect to their distance from the generation origin. Initially I tried with a sorted array but later replaced it with the former data structure to speed up the sorting process. It is not implemented with the intention of having a first-in-first-out solution to objects with identical priority.

To tackle issues linked to generation, the manager uses two radii (properties of the player object) that represent two circle areas around the viewer which are both adjustable. They are used to produce coordinate points around the player which may be the same as chunk objects already present in the environment. The coordinates act as hash values that are used to access and find chunk objects inside several dictionaries and provide a performance improvement because full iteration is avoided, which will be explained in the implementation of the world manager.

The near view distance map

As mentioned before, one view distance is used to keep an area around the player constantly generated. This is achieved by generating points around an origin on a regular grid, the origin having coordinates (0, 0). The generated points form a virtual square grid of size **r * 2 + 1**. This virtual grid spans from (-r, -r) to (r, r) where **r** is the radius (the near view distance). The points are then sorted by distance in relation to the origin and the ones outside the view distance (towards the corners of the virtual grid) are removed.

The resulting map is used as a list of offset coordinates. The world manager first translates this map to the player's position and then it iterates over the translated points (coordinates) to determine which chunks need to be kept generated and rendered on the regular 2D world grid.

The far view distance map

Like the near view distance, the far view distance is used to precompute (in the same way) a map of points residing in the area of a circle with radius being the far view distance. The world manager makes use of this map by translating it to the player's position and then iterating over it to determine which chunks need to be kept in memory and which need to be discarded.

The two maps are used in conjunction with the position of the player. This happens during the first of three passes of which the generation process is comprised. A hash value is used during the passes and is computed by passing an X and Y coordinate. In this sense, I use it by passing X and Z coordinates of points on the regular 2D grid which chunk objects occupy. By specifying any arbitrary point, I can get the respective chunk for it, if it exists already in memory, using its hash value. The world manager must keep separate queues for the chunk that:

1. need to be generated
2. are currently being generated
3. are generated
4. need removal

Generate pass

The first pass (the generate pass) would take the position of the player and it would take a precomputed map of points inside a circle of radius the player's "near view distance". The player position coordinates will be converted into chunk space first and then the precomputed map will be offset by the player position.

The resulting points will be the coordinates for individual chunks and will be used to check against the several queues inside the manager for chunks using the coordinates as hash values. If chunks are currently being generated, are already enqueued for generation or have been generated, nothing should be done. Only if the chunk is marked for removal should it be moved to the generation queue (if not generated) or to the idle queue (if it is generated).

Here I have decided that chunks should be ensured they are always instantiated in the Unity environment. A mesh is currently present for each game object inside Unity which has been instantiated and is used to visualize the chunks before they have had their terrain data be generated. It helps see the overall picture of how the system outputs results.

Remove pass

The second pass of the generation process is the remove pass. During this process, chunks are evaluated in terms of whether they should still be kept in memory. Here the precomputed map for the player's "far view distance" is used. The points it contains are iterated over and are translated according to the player's position on the chunk grid. The pass ensures that all chunks whose coordinate is present in the list of points (in the translated map) are in their correct queues and not in the *remove* queue.

If a chunk that no longer needs to be held in memory, it should be queued to the remove queue. However, it should be noted that no elements residing in the *busy* queue (chunks being generated by threads) are marked for removal as the threaded process will be canceled and for stability reason is avoided.

Build pass

The last pass is the build pass where the queue of chunks that should be generated is built into a priority queue. It is in turn used by the world manager to start individual tasks for chunks to be generated.

Noise Generator

The world manager contains a reference to a noise generator. It is provided to the chunks for them to use to generate a heightmap following a fractal approach. The noise generator is configured accordingly to address this.

3.6 The Unity environment

For the system to functional effectively, Unity also needs to be configured in accordance with the rest of the system. There are several elements that need to be present in a Unity scene for them to work correctly. Some elements come with preset components, but I will only address the ones relevant to my system.

The chunk prefab

A prefab (a game object blueprint) to represent a chunk segment should be first be created. The prefab must have attached to it a chunk instance script, a Mesh Filter component and a Mesh Renderer component.

The Mesh Filter is the Unity component that internally holds the geometry data of a mesh (vertices, triangles, surface normals) and is rendered by the Mesh Renderer component. The pre-generated geometry data should be sent to the Mesh Filter for it to be rendered inside the game engine and as such a reference to the Mesh Filter component must be present inside the chunk instance script.

The camera

The "Main Camera" is a default camera object that comes with every Unity scene. Unity uses it to define an object in 3D space with a certain orientation and having rendering parameters such as (frustum, near and far clipping planes and more) all of which define how the virtual world needs to be rendered on-screen.

The camera alone does not move but instead it follows a player object instead. This is the general approach used by the Unity community and there are even official tutorials that showcase this usage. For this purpose, the camera needs to have a script attached to it that follows a player game object, which is responsible for the movement inside the world. In this solution the script is called CameraFollow.

The player controller

For the camera to follow a moving object, a game object in the Unity scene needs to be placed that would represent the player. A player controller script is attached to the player game object and is used to take player input and move the game object inside the Unity world.

The script enables forward, backward, left, right, up and down directional movement and it also provides an interface for the player to be able to rotate the camera in an FPS style. Additionally, it makes use of the event programming functionality in C# to make a notification method that is called when the player passes several consecutive chunk borders. Only then is generation updated.

The world manager

In the scene should be present a world manager. This is, as with the player controller, a game object on which a world manager script is attached.

The script should hold a reference to a chunk prefab which will be used to instantiate chunks when necessary. Additionally, the world manager needs a reference to the player controller to use as a moving generation origin.

4 Implementation

Having described the system with each of the individual components and how they work, in this section a more technical description including code samples is provided. The idea is to present specific code bits without which the system will hardly be able to operate.

4.1 The Unity environment

Because Unity is the environment in which the project should reside, it is important to make sure our system is compatible with Unity. To do so the scene should be setup in a way to accommodate a Chunk prefab to be instantiated for each chunk segment.

Chunk prefab

A chunk instance script needs to be attached to a Chunk prefab. A Mesh Filter and Mesh Renderer need to be present to render the geometry of the chunk segment. A reference to the Mesh Filter is setup and used externally by the wrapping chunk segment object holding a reference to the chunk instance script.

When told to be destroyed, the chunk instance calls Unity's Destroy(gameObject) method which gracefully handles the object removal from Unity's scene.

The player controller

The player controller script is what Unity uses to intercept user input used in moving and rotating the player game object mentioned before. The WASD keys are used for forward, left, backward and right movement. E and Q are used for upward and downward movement and the mouse will move the camera view horizontally and vertically. Note that the camera is controlled separately by its own script.

The player script also keeps track of the player in terms of how many borders. IT does so using a reference to the coordinates of the last chunk the player was in when generation was updated. The script uses this to determine how far away the player is from the generation chunk and if they have passed n number of chunk borders.

This is the Update() loop of the player controller to fire the border event.

```
int xChunk = ChunkCoordinates2D.ConvertSpace((int)transform.position.x,
    CoordinateSpace.World);
int zChunk = ChunkCoordinates2D.ConvertSpace((int)transform.position.z,
    CoordinateSpace.World);

if (Math.Abs(this.xOldChunk - xChunk) >= borderCrossEventBorderCount ||
    Math.Abs(this.zOldChunk - zChunk) >= borderCrossEventBorderCount) {
    xOldChunk = xChunk;
    zOldChunk = zChunk;
    _onChunkBorderCrossed.Invoke();
}
```

The _onChunkBorderCrossed object is a C# event I wrote below which can be accessed publicly via the OnChunkBorderCrossed public event. The number of borders to cross I specified to be 2, meaning that the player will have enough room to move around before world generation is issued again.

```
private event OnChunkBorderCrossedHandler _onChunkBorderCrossed;
public event OnChunkBorderCrossedHandler OnChunkBorderCrossed {
    add {
        _onChunkBorderCrossed += value;
    }
    remove {
        _onChunkBorderCrossed -= value;
    }
}
```

The camera controller

The main camera game object in the Unity scene needs to have its own script called the CameraFollow script. All the script needs is the positional information for a player game object. As such it uses a public variable inside the script for a Transform component of the player present in the scene.

There are only two lines in the LateUpdate() method used to orient and position the camera. The followedObject variable is the linked Transform component of the player inside the Unity editor for the camera script.

```
transform.position = followedObject.position;
transform.rotation = followedObject.rotation;
```

The world manager object

A game object with a respective script for the game manager is used. The script is called a ChunkManager in the solution. It has a reference to the chunk prefab, a reference to the player's PlayerController script and Transform components setup via Unity's editor. The world manager is separated in its own section and will be discussed implementation-wise later.

4.2 Noise class

The noise class is the most appropriate place to begin with as the whole generation process is fundamentally based on it. It has one constructor that can take up to 5 parameters - the scale, the seed, number of octaves and lacunarity and persistence factors. C# defaulting is used inside the constructor's signature to enable a potential designer who uses this system to choose which parameters are of interest to them. Below is shown the constructor of the class.

```
public NoiseGenerator(float scale = 1.0f,
    int seed = -1,
    int octaveCount = 1,
    float lacunarity = 2f,
    float persistence = 0.5f) { … }
```

Because a PRNG is needed for reproducibility, an instance of System.Random is used. If a seed value is provided, the PRNG is instantiated using with it. I have defaulted the seed to be -1, because I use that number to check if the seed has been manually specified or not. If it has not been specified manually, the PRNG is instantiated using the default constructor.

The PRNG is then used to produce two random values that determine a random value between 100000 and 2000000. The numbers are arbitrarily chosen to be large and are used to move the generation origin far away from the (0,0) point. I wanted to do this because when generating around the (0,0) point resulted in visual mirroring introduced by the Perlin noise implementation. They random values are the generation origin's coordinates and if a world seed is used, they will always be the same.

Below is the code for the Generate method which shows how the input, output and intermediate values are computed using the associated with the generator properties and factors. The call to Unity's scripting API for the Perlin noise algorithm is also present.

```
public float Generate(float x, float y) {
    float frequency = 1;
    float amplitude = 1;
    float value = 0;

    for (int i = 0; i < this.octaveCount; i++) {
        float sampleX = (this.origin.x + x) / this.scale * frequency;
        float sampleY = (this.origin.y + y) / this.scale * frequency;
        float sampleValue = (Mathf.PerlinNoise(sampleX, sampleY) - 1)
            * amplitude + 1;
        value += Mathf.Clamp(sampleValue, 0, 1);
        frequency *= this.lacunarity;
        amplitude *= this.persistence;
    }

    return value / this.octaveCount;
}
```

While I was working on the program, there was a point at which I saw mesh holes occurring during the terrain generation process, such as the one visible in Figure 5 – . I made sure to use the same seed value to reproduce the terrain to later see if my possible fix is correct, although I did not take note of which value was used and cannot provide it here.

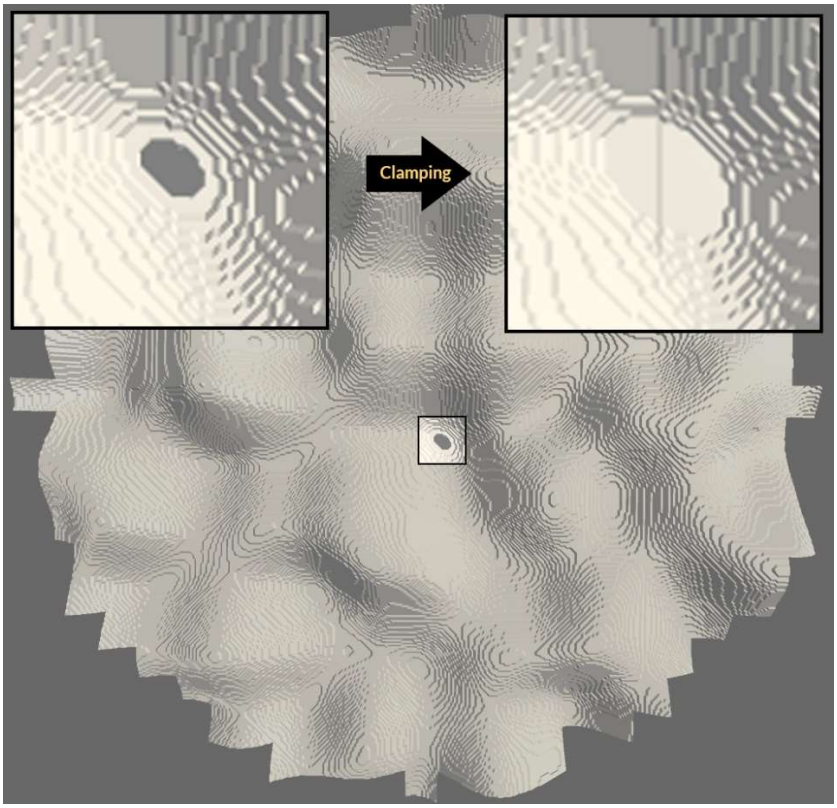


Figure 5 – Mesh hole caused by the Perlin noise implementation used. Fixed by clamping the output noise values between 0.0 and 1.0 inclusive.

The problem of such mesh holes appearing turned out to be because of Unity's Perlin noise implementation. In the documentation (Unity3D, 2018) it is said that the algorithm returns values between 0.0 and 1.0 and in general this is correct. However, there are issues where the algorithm returns values above 1.0 and below 0.0 which is not accounted for and results in inaccuracies. This issue is known and noted by the Unity development team and I have fixed it in my solution by clamping the result of the Perlin noise function to be between 0 and 1 always.

4.3 Voxel object

The voxel is the primitive of the world grid and is used in Marching cubes to extract the respective mesh data. In world coordinates, a voxel's dimensions are 1x1x1 and a private byte data type is used to represent the voxel cell's configuration. Each bit of this byte is associated with a corner of the voxel cell. The index of a corner for each bit is determined by the convention followed, which we have chosen and is shown in 13.

A method that operates on the private byte on the level of individual bits exists. It takes as input parameters the corner index (between 0 and 7, enforced via exception throwing) and a Boolean *state* parameter. The most valuable part of the method is shown below.

```
if (state) {
    this.value |= (byte)(1 << corner);
} else {
    this.value &= (byte)~(1 << corner);
}
```

The configuration byte of the voxel is operated on using bitwise operators because on the lowest level it is faster to use them instead of doing calculations. Each corner is associated with one bit where the index of the corner is treated as the position of the bit counting from the least to most significant bit.

I had some issues with implementing the coordinate offsets for the edges of a voxel inside Marching cubes, so instead I created a helper method for getting the coordinate offset from the origin point to the middle of each edge. The offsets are hardcoded for each edge and the origin point overlaps with the position of corner index 0. It takes as input an edge index and return a 3D offset vector.

An example is provided for the first two cases, “default” referring to edge index 0. The edge value is also checked, and an exception is thrown if the index is invalid. It helps with the correctness of the program.

```
switch(edge) {
    default:
       return new Vector3(0.5f, 0f, 0f);
    case 1:
       return new Vector3(1f, 0f, 0.5f);
```

4.4 Coordinates

The 2D coordinates class has X and Y private values associated with it. The values are classified as readonly because individual chunk objects will never have a positional change in the current implementation, so for optimization purposes this is preferred.

I have written a base constructor that takes two coordinates and an enumeration representing the space. The constructor then, depending on the space in which the input coordinates are, makes sure that the final representation of the coordinates is in chunk space.

The coordinate space enumeration

```
public enum CoordinateSpace {
       Chunk,
       World
}
```

The 2D coordinates class constructor

```
public ChunkCoordinates2D(int x, int z, CoordinateSpace space) {
    if (space == CoordinateSpace.World) {
       this.x = x >> 4;
       this.z = z >> 4;
    } else {
       this.x = x;
       this.z = z;
    }
}
```

I have also written a static helper method that allows the conversion of values from one space to another without having to access a specific instance of the coordinates class. This was useful in accordance with the world manager because, as mentioned in the previous section, it uses two precomputed maps of translated coordinate points to determine which chunks need to be generated and kept in memory. The manager needs to convert the points from chunk space into world space and the static method enables this.

4.5 Chunk and chunk segments

The chunk is a container for a column of voxels and its dimensions have been chosen to be 16 units along the X and Z axis. Because Unity introduces a hard limitation where each game object can have a mesh of at most 65000 vertices, chunk segments are used to enforce the compatibility of the generated mesh data with Unity and they are of size 16 units in all three axes.

4.5.1 Chunk segment

Chunk segments must store a reference to a script object which inherits from Unity's MonoBevahior (scripting) environment to be able to interface with Unity. In the project solution this script object is called a ChunkInstance and as such a private ChunkInstance reference is included.

There is a method which instantiates the ChunkInstance script linked to the specific chunk segment by creating a game object in Unity using a prefab. The chunk segment has with it associated coordinates. The method in question takes as input this prefab object, which is defined in the world manager, and is passed to the chunk segment through the chunk encapsulating it.

To instantiate a game object and link it with the chunk segment I am using the following code:

```
public void CreateChunkInstance(GameObject prefab) {
    if (this.chunkInstance == null) {
        this.chunkInstance = GameObject.Instantiate(prefab,
        this.coordinates.GetXYZ(CoordinateSpace.World),
        new Quaternion(0, 0, 0, 1)).GetComponent<ChunkInstance>();
    }
}
```

Voxels are stored in the chunk segment inside a 1D array and whenever it is accessed, the coordinate of a specified voxel with position **x**, **y**, **z** will result in the value at location **x + z * 16 + y * 256** (chunk segments are 16 units in dimensions).

The chunk segment also provides a method to apply the stored in it mesh data to the ChunkInstance reference it holds. Without the following lines of code, the mesh data contained in the object can never be rendered by Unity.

```
public void ApplyMesh() {
    Mesh mesh = new Mesh();
    mesh.Clear();
    mesh.vertices = this.meshData.Vertices;
    mesh.normals = this.meshData.Normals;
    mesh.triangles = this.meshData.Triangles.ToArray();
    this.chunkInstance.meshFilter.mesh = mesh;
}
```

A destroy method is also required for cleanup purposes when the object is no longer needed in memory.

```
public void Destroy() {
    this.chunkInstance.DestroyInstance();
}
```

4.5.2 Chunk object

The chunk object is the parent of individual chunk segments. The chunk object defines a *height* which is used to determine how many chunk segments are stacked vertically. The height is also used inside the chunk object to upscale the elevation values produced after running the supplied (by the world manager) noise generator instance. The chunk segments are stored in an array of size height. The class also has a method that iterates over the chunk segments and calls their CreateChunkInstance (GameObject prefab) method by passing a reference to the prefab which is given by the chunk manager to a wrapper method called CreateInstances (GameObject prefab) inside the chunk class. It also has a method called GenerateTerrainAndMarch () which computes calculates the respective noise values and populates them first into a heightmap which is then used to create fill the 3D volume of the specific chunk object.

Chunk hash code

A hash code is also computed for the chunk object by using the 2D chunk coordinates which are 32-bit values each to form a 64-bit number. The implementation of the hash code calculation is the following:

```
public ulong GetChunkHashCode() {
       return (ulong)coordinates.GetX(CoordinateSpace.World) +
           ((ulong)coordinates.GetZ(CoordinateSpace.World) << 32);
}
```

The chunk hash code is generated every time the chunk object is instantiated. The key and chunk object associated with it are inserted into a global hashmap structure which keeps track of all instantiated chunk objects in the game world. This data structure is needed internally for the chunk object to provide a simple "Getter" method for a chunk. It takes x and z values alongside the space in which the coordinates are and using the same function computes a hash. It is used to try and pull a chunk reference for a live chunk object. If it fails to get an existing chunk, it returns a new empty chunk object. It is also important to ensure the input coordinates are in world space, else there will be clashing keys because in chunk space 0 to 15 will always result in a 0 (bit shifting to the right by 4 loses the first 4 bits), so I used the static helper method in my 2D chunk coordinate class.

Heightmap

For the generation process, a heightmap is constructed inside the chunk from which voxel grids are constructed and piped to individual chunk segments. The construction of the heightmap has the following code:

```
float[] heightmap = new float[17 * 17];

for (int y = 0; y < 17; y++) {
    for (int x = 0; x < 17; x++) {
       heightmap[x + y * 17] = noisegen.Generate(
           x + coordinates.GetX(CoordinateSpace.World),
           y + coordinates.GetZ(CoordinateSpace.World)) * 16 * height + 1;
    }
}
```

Note that I am using the number 17 because I am populating a voxel grid with dimensions being 16. No matter how many voxels are stacked consecutively, the total number of values in one row of a grid of corner values will be equal to the number of voxels plus one. Also remember the height value is specified in the object and is known to all its instances.

Voxel grid generation

For each chunk segment in the chunk, an array of 4096 is used to store individual Voxel objects. Three loops are iterated over along the Y, Z and X values following my chosen format for the voxel grid. I decided that voxels will be generated first along the X axis to form a row, then multiple rows will be generated along the Z axis to form a slice which will finally be stacked vertically to form a complete chunk.

When each cell is examined, its corner values need to be determined. The heightmap is used for this purpose. Each grid cell on the heightmap (it is a 17x17 grid) is used to determine how high along the Y axis a column of voxel corner values is filled. Because the heightmap is already scaled, only the height offset of individual chunk segments needs to be obtained from the chunk segments. Then relative to it, series of corners along the Y axis can be set to 1 until the specified by the heightmap height value is surpassed.

Below is the code executed by each iteration of the three stacked loops:

```
Voxel voxel = new Voxel();
voxel.SetCorner(0, heightOffset + y <= heightmap[x + z * 17]);
voxel.SetCorner(1, heightOffset + y <= heightmap[x + 1 + z * 17]);
voxel.SetCorner(2, heightOffset + y <= heightmap[x + 1 + (z + 1) * 17]);
voxel.SetCorner(3, heightOffset + y <= heightmap[x + (z + 1) * 17]);
voxel.SetCorner(4, heightOffset + y + 1 <= heightmap[x + z * 17]);
voxel.SetCorner(5, heightOffset + y + 1 <= heightmap[x + 1 + z * 17]);
voxel.SetCorner(6, heightOffset + y + 1 <= heightmap[x + 1 + (z + 1) * 17]);
voxel.SetCorner(7, heightOffset + y + 1 <= heightmap[x + (z + 1) * 17]);
voxels[x + z * 16 + y * 256] = voxel;
```

"voxels" is the array storing 4096 voxels for each chunk segment. After the data is stored, it can be sent to the specific Chunk Segment via a method to set the data. I use a simple SetData(Voxel[] voxel_array) setter method.

Destroy method

The chunk object's Destroy() method is called by the world manager when the chunk needs to be destroyed. Its purpose is to relay the manager's order to all individual chunk segments so that they can tell Unity that their associated game objects are no longer needed and should be destroyed. It is finally removed from the global array and will not be used in any calculations by the world manager.

```
public void Destroy() {
    for (int i = 0; i < this.chunkSegments.Count; i++) {
       this.chunkSegments[i].Destroy();
    }

    this.chunkSegments.Clear();
    chunks.Remove(this.hashcode);
}
```

4.6 World manager

The world manager is probably the most complicated module of the system. It is responsible for the management of the whole world and what resides in it and as such uses several methods, data structures and namespaces to enable this. It uses a priority queue to keep an always sorted (according to distance from player) data structure that outputs chunk objects which are of lowest priority (distance from player).

Tasks

It uses C#'s threading pool capabilities using the System.Threading.Tasks namespace. In the program, tasks are objects which are associated with one chunk object each and at any time there can be at most **n** number of tasks.

The need for putting a cap on the tasks was because once a task is created and started, it will be pooled into C#'s internal thread pool. Interrupting the task is possible although to ensure the safety and stability of the program, because I am not familiar enough with tasks, a limit was used to have at most a small subset of tasks running. If they are no longer present in the viewable by the player area, then once

they are generated and the border crossed event is fired (as talked before), the chunks will be queued for removal.

A task is created like this:

```
Task<Chunk> task = new Task<Chunk>(() => {
    chunk.GenerateTerrainAndMarch(noisegen);
    return chunk;
});
```

The chunk object comes from the priority queue implemented by calling its Dequeue () method. The passed *noisegen* argument is a reference to an instantiated NoiseGenerator class to be used by the function when generating the chunk heightmap. It is by default configured to use 3 octaves, the results of which are summed and then divided by the number of octaves (i.e. 3). Having less octaves makes the terrain look duller while having more makes it more realistic, although it also introduces performance degradation.

After creation, the task is added to a list of currently created tasks and is started. The list is traversed by the world manager in the Update () function described further below.

The distance function

I use a simple function to decide the distance between two points. For a point *a* with coordinates *x, y* and a point *b* with coordinates *u, v* the mathematical formula I used for computing the distance is $(u - x)^2 + (v - u)^2$. The proper formula is to *sqrt* the result of the previous equation, but I do not use that because I achieve the same relative priority for a chunk with a given point by using the squared result instead. This saves the CPU some unneeded cycles, which is important considering the distance function is used during the generation process.

Precomputing the maps

To compute the near and far view distance maps, I made a method that takes a radius as input and outputs a 1D array of sorted points. I am using a C# List<> object to first add to and then later remove points from. The array is populated with this code:

```
for (int x = -radius; x <= radius; x++) {
    for (int y = -radius; y <= radius; y++) {
       points.Add(new Tuple<int, int>(x, y));
    }
}
```

The sorting method I used is this, where *origin* is a tuple of (0, 0) and points being the array of generated points from the previous code:

```
points.Sort(delegate (Tuple<int, int> p1, Tuple<int, int> p2) {
    double d1 = Distance(origin, p1);
    double d2 = Distance(origin, p2);
    return d1.CompareTo(d2);
});
```

The generation process

First the event defined by the player controller, which is fired after several chunk borders are crossed, needs to be linked with a function to call inside the world manager class. I have done this with the line show below. Note that the playerController variable is a reference to the player object's playerController script in the Unity scene and UpdateChunkGeneration () is the method to call when the event is fired:

```
this.playerController.OnChunkBorderCrossed +=
    new OnChunkBorderCrossedHandler(UpdateChunkGeneration);
```

Because the world manager needs to have 4 buckets for chunks to be put in, I used 6 Dictionary objects, 2 of which are used as buffers. I chose to use dictionaries because when I iterate over the precomputed maps, I would get final coordinates for the points which need to be generated and kept in memory. I can directly use the points' coordinates by turning them into a hash and then checking the dictionaries for the chunk objects.

The buffer objects are used when populating the generate and done chunk dictionaries during the remove pass. Because I only have the points that are of interest, I need to take out the chunks that should be kept and leave only the chunks that should be removed. By removing chunks from the dictionaries and placing them in the buffers, I can safely iterate over the left-over chunks and mark them for removal. Then I swap the contents of the *generate* and *done* dictionaries with the contents of their respective buffers. An example of how a dictionary was setup is shown below:

```
private Dictionary<ulong, Chunk> c_generate;

void Start() {
    c_generate = new Dictionary<ulong, Chunk>();
}
```

Buffers decrease performance issues by removing the need to iterate over the complete list of chunks, to determine which ones need to be removed, by using a precomputed map to separate the chunks that should be kept. I had some issues using the buffer dictionaries initially in the remove pass. I needed to iterate over the keys which are left in the *generate* and *done* dictionaries to mark all the chunk objects for removal, but in C# it is not possible to be iterating over a collection while at the same time removing objects from the iterated over collection. Instead, I iterate over the keys in the *generate* or *done* dictionaries (after the items that should be kept have been moved to the buffers), add the left-over chunk objects to the remove dictionary and store a reference to the key in a separate *keybuffer* array of long values (the hash codes, or keys, of the chunks used in the dictionaries). I finally iterate over the keybuffer and remove the items from the corresponding container.

The BuildPass () function takes as input parameters the positional offset in chunk coordinate space and in it constructs a priority queue. It is the third pass of the three generation passes and is necessary to provide a correct representation of the priorities of individual chunk objects. The priority queue and its node are discussed in the next section.

```
private void BuildPass(int offset_x, int offset_z) {
    this.chunkQueue.Clear();

    foreach (Chunk chunk in this.c_generate.Values) {
       PriorityQueueNode<Chunk> node = GetNode(chunk);
       this.chunkQueue.Enqueue(node,
           (int) Distance(new Tuple<int, int>(offset_x, offset_z),
       new Tuple<int, int>(chunk.coordinates.GetX(CoordinateSpace.Chunk),
       chunk.coordinates.GetZ(CoordinateSpace.Chunk))));
    }
}
```

The update method

The update method is where orders are issued. The removal of chunks (if any are marked for removal) and updating of tasks used for chunk generation are performed here.

For chunk removal, all I do is to have a timer which keeps track of the time. When a certain amount of time passes, a limited number of chunks is destroyed by calling the chunk's Destroy () method which is transferred to all chunk segments and resultantly to the game objects inside Unity. The timer then resets. I have decided to use a timer of 0.25 seconds and at most 16 chunks will be destroyed in that time. Balancing this should be considered because during the process of removal, Unity introduces a somewhat large overhead for destroying game objects.

Tasks were previously mentioned to be used in generation. When a task is started, it is enqueued to be processed by a thread in C#'s internally managed thread pool. The Update () method first iterates over the list of currently active tasks and determines if a task has finished. If it has, the chunk reference contained in the task is used to order the chunk to update its mesh which is ultimately reflected in Unity. The task is then removed from the list allowing for the next chunk to be added and processed.

The method then checks if the maximum number of tasks concurrently allowed has been reached. If the limit has not been reached, a new task is created, started and added to the list of tasks. The number of maximum tasks does not necessarily impact performance, but it is used to limit the number of chunks that are "busy" or are being generated. Remind yourself that when chunks are busy being generated, they cannot be moved between queues. Therefore, if during generation there are tasks inside the task list, the respective chunks will be generated even if they are outside the outer view distance. They will be removed the next time the generation process is ran as long as they are still outside the far view distance.

4.7 Priority queue

The priority queue implementation has been largely based on an interface which I obtained from (BlueRaja, 2017). The node class is very simple and has been implemented as such:

```
public class PriorityQueueNode<T> {
    public int priority;
    public int index;
    public T data;

    public PriorityQueueNode(T data) {
       this.data = data;
    }
}
```

The priority queue internally has a list of nodes, a *max nodes* count to initialize it and an index counter which points to the next free location in the array where a Node can be inserted. The priority queue is based on an array to store the nodes. Since distance is used as the priority of a chunk and the closest non-generated chunk should be the next one to generate, the first node to always be dequeued should be of shortest distance and thus of lowest priority.

The most essential methods of the Priority queue are the Swap(), Enqueue() and Dequeue() methods.

Swap method pseudocode

Swap(a, b)
a – the first node to swap
b – the second node to swap
ai – the first node index
bi – the second node index
n – The global nodes array

1. create *temp* variable
2. store *ai* in *temp*
3. store *bi* in *ai*
4. store *temp* in *bi*
5. store *a* in *n[ai]*
6. store *b* in *n[bi]*

Enqueue method code

```
// Place the new element in the next available position in the array
node.index = _currentIndex;
node.priority = priority;
nodes[_currentIndex] = node;

// Compare the new element with its parent - if smaller, then swap with parent
PriorityQueueNode<TData> parent;
while ((parent = GetParent(node)) != null && parent.priority > node.priority) {
    Swap(node, parent);
}

_currentIndex++;
_nodeCount++;
```

The GetParent() function returns the node located at nodes[node.index / 2].

Dequeue method code

```
PriorityQueueNode<TData> dequeuedNode = nodes[1];
PriorityQueueNode<TData> swapperNode = nodes[_currentIndex - 1];
nodes[1] = swapperNode;
nodes[_currentIndex - 1] = null;
swapperNode.index = 1;

PriorityQueueNode<TData> left = GetLeft(swapperNode);
PriorityQueueNode<TData> right = GetRight(swapperNode);

while ((left != null && swapperNode.priority > left.priority) ||
    (right != null && swapperNode.priority > right.priority)) {
    if (left == null) {
       Swap(swapperNode, right);
    } else if (right == null) {
       Swap(swapperNode, left);
    } else {
       if (left.priority <= right.priority) {
           Swap(swapperNode, left);
       } else {
           Swap(swapperNode, right);
       }
    }

left = GetLeft(swapperNode);
right = GetRight(swapperNode);
}

_currentIndex--;
_nodeCount--;
return dequeuedNode;
```

The GetLeft() function returns the node located at nodes[node.index * 2] as long as the index is within array bounds. The GetRight() function returns the node located at nodes[node.index * 2 + 1] also as long as the index is within array bounds.

4.8 Marching cubes

Marching cubes works on individual voxel cells. Depending on their configuration, a mesh will be constructed according to a concrete specification present in a triangle table or *triTable*. This table is large it would have been unfeasible for me to do this manually or write separate code just to compute the values. Instead, I used the table provided in (Bourke, 1994), which was the source I followed by large to produce my implementation of Marching cubes.

The first issue I came across when implementing Marching cubes was how big the arrays for storing vertices, triangles and normal should be. The format in which Unity needs the mesh data to be is for vertices and normal to use a simple array structure. The triangles are stored in a List<> however. The calculation I derived (and previously mentioned in the report) is the dimensions of the grid (16) to the power of three, times 5 (maximum number of polygons) and then multiplied by 3 (3 vertices per triangle).

Because I was going for a flat style, the second issue I encountered was that I was unsure how to change the implementation provided by Paul Bourke. I did further research on my idea and debugged the solution continuously to see how correct my surface normal calculations are. In the end, instead of using 1 normal for the same vertex which would almost always be shared by multiple triangles, I had to assign the same surface normal to each of the three vertices for each triangle. Below is the code I wrote which deals with iterating through the data from the triangle table using the X, Y and Z of the voxels. The order in which voxels are traversed and the for loops are nested is Y, Z, X. I had to use a counter *v* to keep track of the next free position in the vertex array.

```
for (int i = 0; triTable[voxelValue, i] != -1; i += 3) {
    Vector3 vertex_0 = voxelCoordinate +
        voxel.GetEdgeOffset(triTable[voxelValue, i]);
    Vector3 vertex_1 = voxelCoordinate +
        voxel.GetEdgeOffset(triTable[voxelValue, i + 1]);
    Vector3 vertex_2 = voxelCoordinate +
        voxel.GetEdgeOffset(triTable[voxelValue, i + 2]);
    Vector3 normal = Vector3.Cross(vertex_1 - vertex_0, vertex_2 - vertex_1);
    vertices[v] = vertex_0;
    vertices[v + 1] = vertex_1;
    vertices[v + 2] = vertex_2;

    triangles.Add(v);
    triangles.Add(v + 1);
    triangles.Add(v + 2);

    // Assign identical normals to achieve flat artistic style - 1 normal per vertex
    normals[v] = normals[v + 1] = normals[v + 2] = normal;

    v += 3;
}
```

5 Results and Evaluation

In this section I will provide some general details about the results of the generation as well as some level of evaluation in terms of the performance of individual elements. Due to time constraints I was not able to perform more tests on the program. For example, I have not tested the performance changes introduced by concurrently scheduling different numbers of tasks. Another example was that I did not test the environment with limited memory capacity to assess how the generation (and overall program) perform when the limit is exceeded.

5.1 Priority Queue

5.1.1 Unit Testing

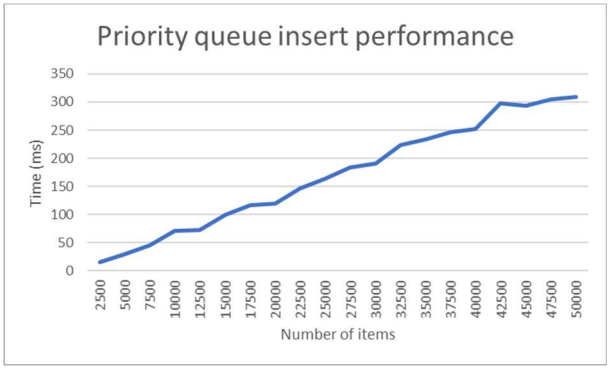
Unit tests are essential parts of building a system which is prone to changes. For this project, a priority queue was used and to ensure the correctness of the implementation of methods, a set of unit tests for the major methods of the priority queue were written and used to debug issues that arose with its implementation.

The way the testing environment was setup was sufficient to implement and test the priority queue, although the way the tests were implemented was probably wrong. They were situated in a separate project from the main which was able to “see” the namespace in which the priority queue was included. The tests would call the tested methods and then using class properties and getter methods, values were compared against what the expected output was.

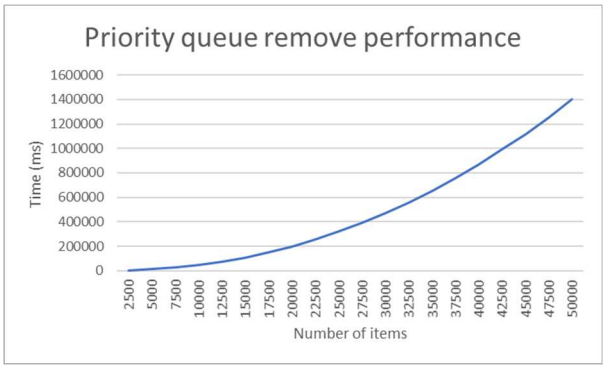
The proper way to write the tests was to include them the priority queue class and then compare the internal state of the data structure with what is expected.

5.1.2 Insert and remove performance

For the correctness of the priority queue, I measured its performance for the insert and pop methods respectively. The priority queue was populated with 2500 elements, counting to 50000 with increments of 2500. This was done 50 times and the results have been averaged. The produced results are plotted and shown below.



(a)



(b)

In diagram (a) we can observe a time complexity of about O(n) and for removal (b) the complexity appears to be O(N logN). The results of this are expected if an array is used as the backbone of the priority queue. If a heap is properly implemented, insertion and removal can be achieved using O(logN).

5.2 Terrain generation

Terrain generation is not perfect in this program and suffers in one way or another. Below are detailed individual issues that were spotted with terrain generation.

5.2.1 Visual stutters when chunks are instantiated

Currently, every time the generation process is executed, a noticeable stutter may sometimes be present. To find out why this happens, I used Unity's Profiler and enabled the "Deep Profile" option so that it can time methods outside the Unity environment as well.

I ran the generation process and waited for the whole area to finish generating. Then, I moved sideways a bit until I crossed 2 chunk borders, after which the world manager ran the generation process again. I waited a bit and paused to examine the performance.

There was a visible spike the moment I crossed the two chunk borders and by examining the method calls I saw that the primary reason for the drop in performance was the GameObject.Instantiate() method by Unity, which means that Unity itself introduces quite a significant overhead for this task.

5.2.2 Generation artifacts

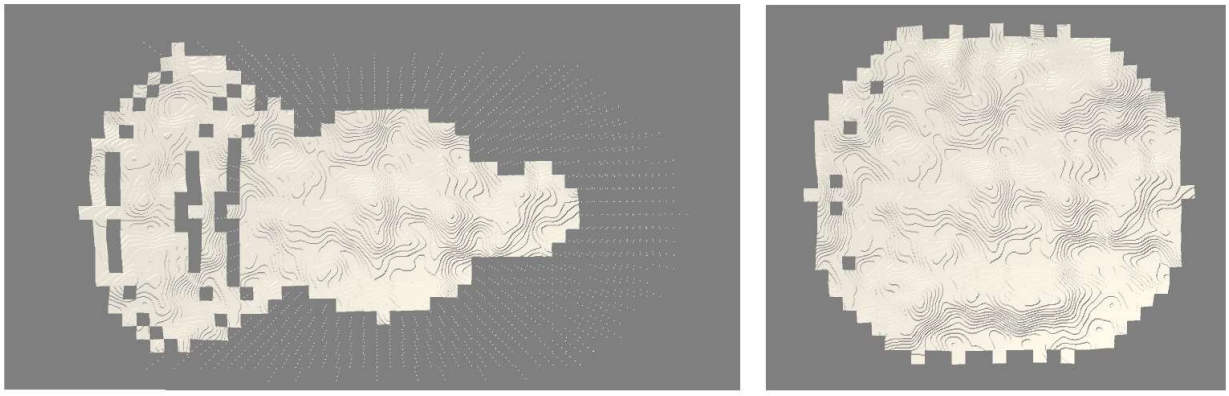


Figure 6 - Visible "radial" artifacts present after moving right and then a bit left

In the images above on the left-most side of the generated area can be seen holes. The left image shows clear radial bands which I believe is due to the way dictionaries work internally in C#. When the generation is updated, chunks outside the larger view distance should be marked for removal. As detailed before, this is done by removing the chunks that should be kept into a buffer and then iterating over the left-over elements and moving them to the *remove* dictionary. The actual chunk removal is a simple iteration over the values contained in the remove dictionary and as such the order in which chunks are stored is governed by the way dictionaries are implemented in C#.

The artifacts present on the left side of the right picture are because of the player moving to one side and then going back. When the player moves in a direction, chunks behind them will get dropped

after a certain radius. When chunks are marked for removal, the program immediately starts to remove chunks on a per-quarter second intervals. If the player is fast enough to come back, there will be chunks that would not have been removed yet and will be preserved. This happens as the player's large view distance area was once overlapping the chunks, then it stopped overlapping them and then it came back and started overlapping them.

The outer view distance does not enforce generation to take place and as such the holes are not going to be "fixed" until the player gets close enough for the coordinates of the missing chunks to reside in the circle of radius the *near view distance* around the player.

In the context of a game, these artifacts would generally not be a problem. In normal scenarios, a player will not be fast enough to leave an area and then come back to it and have it be partially generated. The location of the artifacts is also towards the outer sides of the region kept in memory and is not too problematic, although a solution can be sought.

6 Future Work

This section is meant to provide ideas about further work that could be carried out to improve the state of the project. Present issues and missing features are addressed, and expandability is also talked about.

6.1 Improve terrain realism

Currently terrain in the project is made using three passes of Perlin noise sampled at different resolutions which produces OK results. The terrain realism aspect can be improved by implementing other 2D and 3D noise generators. Functionality to modify volume data directly (as would be necessary from a 3D noise generator) will need to be implemented inside the chunk object.

Features such as caves and canyons should be sought and produced by using such noise generators. It is important to make sure the values of the new algorithms are within the agreed range of the system.

6.2 Move generation to GPU

The current implementation greatly solves generation performance speed by creating a task for each chunk to be process by a thread. This is done on the CPU and is significantly slower than if done on the GPU. Shaders can be written for the GPU to calculate the 3D grid of voxels for each chunk which will result in a significant boost in performance. The two generation elements that can be parallelized are the noise generators used and the Marching cubes as the former produced a 3D volume of data while the latter uses it.

6.3 Moving the system to a custom engine

Unity is an environment optimized for making all kinds of games. This means that internally, Unity most likely has some elements which are not optimized for the problem solved by this project which introduced visible performance penalties and needed workarounds that limit the capabilities of the system.

It would be better to remove Unity from the list of possible factors that influence the performance of the solution and instead a custom engine which supports rendering should be built and used. By implementing a custom engine there should be more room for optimization because the system would no longer depend on Unity.

7 Conclusions

In summary, this project develops a program that generates terrain based on a provided noise generator. It uses 2D Perlin noise from Unity the results of which are configurable using a world seed, scale, positional offset, octave count, persistence and lacunarity. The world is spatially divided into a 2D grid of chunk objects and is based on fractional Brownian motion.

The whole process of generation has been parallelized and is managed by a world manager which decides the portion of the landscape that needs to be adaptively generated around a moving observer. Terrain generation is done at configurable intervals which are linked to the time it took the player to cross the border of several consecutive chunks.

The procedural generation system is entirely compatible with Unity and all the data is rendered within its scene view.

8 Reflection

The reason I chose the project was largely due to my own interest in procedural landscape generation in the context of video games. I was not very familiar with how such a system would work and what elements I should be careful about in terms of performance, but I had the drive to create my own unique solution. I was determined to work on the project and was certain that despite the lack of knowledge I would be able to familiarize myself well-enough with the different aspects of procedural landscape generation to produce a final working solution.

The final results are in my opinion reasonable, although I would prefer for them to be improved. If I were to play a game which implements my current system I would likely be bothered by the performance issues caused by the generation process happening on the main thread and the garbage collection introduced by C#, which I have little control over due to being tied to using Unity.

For this reason, I have decided that if I were to build a game which heavily focuses on the usage of data structures and algorithms to split up and manage spatial data, I would prefer to write my own engine. This is because during this project I encountered numerous implementation and performance issues with Unity and I realized that for a system to perform on an above average level, it needs to be heavily optimized for the task in question. Unity is not very good at that because it is a game engine that is meant to be overall efficient for building games where the worlds are static.

I have also learned the importance of planning and doing extensive research as well as choosing what types of documents to use for research and implementation purposes. During the initial implementation of the project solution I relied mostly on community answers for solving technical issues. However, while implementing Marching cubes I better understood the significance of using well-

structured and well-written official documents. The detailed solutions present therein are of higher quality and a lot of time can be saved by reading through and understanding the problem and its solution in completeness. From now on I will focus more on using official papers to solve broader problems such as procedural content generation where my knowledge is scarce. I will also aim at solving the problems with the consideration of what is already available, rather than trying to “reinvent the wheel” from scratch.

Technically speaking, I learned how to approach implementing algorithms and data structures by keeping performance and correctness into consideration. I have familiarized myself further with the technicalities of using thread pools and tasks to be processed by threads. I have also gained deeper understanding in the subject of spatial data distribution and iso-surface extraction.

If I am going to be working on another project of a subject I am not very well familiar with, I would put more emphasis on finding and reading relevant papers and books that explain how a similar system would work. Once I have gotten an idea of how a general system would work I would create a design of my own which I will then start implementing. I would also make sure to keep a constant back up of my progress to avoid possible data loss.

9 Bibliography

Archer, T. (2011). *Procedurally Generating Terrain.* Sioux City: Travis Archer.

BlueRaja. (2017). *High Speed Priority Queue for C Sharp*. Retrieved from Github: https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp

Bourke, P. (1994). *Polygonising a scalar field.* Paul Bourke.

Ebert, M. P. (1994). *Texturing and modeling: A Procedural Approach.* Academic Press Inc. 1994.

Gibson, S. F. (1999). *Constrained Elastic SurfaceNets: Generating Smooth Models from Binary Segmented Data.* Cambridge: Mitsubishi Electric Research Laboratories, Inc., 1999.

Ju, T. (n.d.). *Dual Contouring of Hermite Data*. Retrieved from http://www.cs.wustl.edu/~taoju/research/dualContour.pdf

LINGRAND, M. D. (2002). *The Marching Cubes*. Retrieved from http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

Perlin, K. (2001). *Noise hardware. In Real-Time Shading SIGGRAPH Course Notes*. Retrieved from https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf

Perlin, K. (2016, March 3). *Making Noise*. Retrieved from https://web.archive.org/web/20160303232627/http://www.noisemachine.com/talk1/

Persson, M. A. (n.d.). *Minecraft*. Retrieved from https://minecraft.net

Philip M. Sutton, C. D.-W. (1999). *A Case Study of Isosurface Extraction Algorithm Performance.* Amsterdam: Institute of Electrical and Electronics Engineers Incorporated 2000.

Red Blob Games. (2013, August 31). *Noise Functions and Map Generation*. Retrieved from Red Blob Games: https://www.redblobgames.com/articles/noise/introduction.html

Smelik, R. M. (2009). *"A survey of procedural methods for terrain modelling" in Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS)*.

Unity3D. (2018, 04 30). *Unity Manual*. Retrieved from Unity: https://docs.unity3d.com/Manual/index.html

Unity3D. (n.d.). *HTTPS://UNITY3D.COM/*.

William E. Lorensen, H. E. (1987). *MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE COSNTRUCTION ALGORITHM.* New York: General Electric Company.