Procedural Generation of Voxel Worlds with Castles

A thesis

submitted in fulfilment

of the requirements for the degree

of

Master of Science

at

The University of Waikato

by

SEBASTIAN DUSTERWALD

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2015

Abstract

This thesis explores procedurally creating voxel based terrains and creating castles in them. With the explosion of interest in using voxels in games since the success of Minecraft, and even big budget titles like EverQuest Next using this technology, this research aims to address an area that has not received adequate attention: procedural generation of buildings like castles.

By utilizing voxels combined with procedural generation of terrain, games like Minecraft and its many clones are able to offer worlds of near limitless size for the player to explore. However, this presents a challenge to fill this world with content worth exploring. Ultimately this means structures that look human-made. This project focuses on perhaps the most iconic of fantasy game structures: the castle.

A voxel world engine was developed to explore this problem, complete with a procedural terrain generator similar to that used in voxel world games. The terrain generator developed for this project is capable of creating a variety of terrains randomly to test castle placement and generation in.

By drawing inspiration from how real castles were constructed an algorithm was developed to analyze an area of voxel terrain for optimal positioning and a further optimization step was added to adapt the layout of the castle to the terrain. This was used to procedurally generate castles that aim to look like they were placed into the world by a human agent.

Acknowledgements

I would like to thank everyone who helped to make this research and thesis possible. Particular thanks go to the University of Waikato Computer Science department for letting me do this Masters. Also the University of Waikato scholarship trust for providing me with the financial backing that made this possible.

I want to thank Bill Rogers for guiding me through the process of the research and advising me in the writing of this thesis.

I want to thank all the anonymous participants in my user study that helped me to evaluate my work.

Lastly I want to thank my family for supporting me through this adventure.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Volumetric elements (voxels) are the three dimensional equivalent of pixels. The idea is a simple one, dating back to the earliest days of computer graphics research, however the large memory requirements of even fairly coarse scenes represented in voxels has meant that they have historically found only limited application. The primary usage of voxels has been in the medical imaging field, where voxel representations are typically used for MRI scan results and the like. Voxels have also been tried in video games at various times, with mixed success. One of the first uses of voxels was to render the terrain in the 1992 video game Comanche Maximum Overkill (NovaLogic Awarded Patent for Unique 3-D Graphics Engine, 2000) (see Figure 1). Using a ray-casting approach into a 2D heightmap, this allowed for much higher graphical fidelity in the terrain than had otherwise been possible at the time. However, the rise of 3D acceleration hardware, such as the Voodoo add-in board from 3dfx (now defunct (Form 8-K: Bankruptcy or receivership, 2002)), meant that polygon rasterizers advanced by leaps and bounds and interest in alternative rendering methods fell away for many years.

Figure 1- Comanche Maximum Overkill
(Comanche: Maximum Overkill screenshots for DOS - MobyGames)

As the amount of computer memory and processing power available to programs has skyrocketed over the years, the voxel approach has again become more attractive. In computer graphics packages for film and television, such as AutoDesk Maya™, voxels have now been integrated for various volumetric effects workflows, such as cloud and fluid simulation. As such they have been used to produce effects for big budget movies such as Lord of the Rings and The Day After Tomorrow (Crassin, Neyret, Lefebvre, Eisemann, & Sainz, 2009).

Starting with the relatively unknown InfiniMiner and being popularized by the wildly successful Minecraft, voxels have now managed to spawn a whole new genre of video games referred to as "Voxel World" games. In these games the entire world is constructed out of voxel cubes. These voxels are converted to polygons for rendering on modern graphics cards that are optimized for scanline conversion of triangles. A voxel in this case describes a single element in the world, much the way a tile does in a 2D tile-based game such as Super Mario Bros. While voxels are often converted directly into cubes, they can be converted into other shapes (such as torches or stairs in Minecraft for instance).

The appeal of voxel-based games is that they, unlike traditional games, allow for complete manipulation by the player or players. Having a completely destructible world has long been a holy-grail of sorts for computer games. A number of approaches have been tried, usually by simply making some objects destructible and/or by heightmap deformation. All of these methods have been quite limited. Voxels not only make the entire world destructible in a coherent fashion; they also make the world *constructible*. Now the player can actually build anything they want in a voxel sandbox as well. Other games have also tried to give constructive control to the player but the grid-based layout of a voxel world also has a couple of big advantages. Firstly, it puts an absolute constraint on the geometric complexity that can be achieved in any given area. In contrast any approach that lets the player place arbitrary models into the world creates the potential for the creation of scenes too complex to render at satisfactory speed. Secondly, the concept of stacking blocks on each other is very simple and easy to grasp. Hence, even young children have little difficulty in building complex architecture in a game like Minecraft, while other construction methods like CSG (constructive

solid geometry, i.e. Boolean operations) are frequently difficult for even skilled adults to use properly.

This flexibility and simplicity does come at a price: graphically the world is presented at low resolution. Elements are broken up into large chunks, and most of the world is constructed from cubes, typically at a scale of about $1m^3$ relative to the player. Such cubes are usually constructed with face-aligned normals, further enhancing the harsh and blocky appearance of the world. This coarse appearance can put some people off, but the low resolution and blocky nature of these voxel worlds is part of what makes building in them so accessible because it just becomes a matter of placing or removing large blocks.

In order to be able to provide the player with a large world to explore, most such games employ procedural methods for world generation. In fact, by creating new parts of the world on-the-fly, games such as Minecraft are able to provide the player with worlds that are in effect limitless[1] (Persson, 2011). This is a highly desirable feature in such games for several reasons. Firstly, since they are generally sandboxes and centred on building, an infinite world means infinite resources for construction and unlimited space in which to build. Furthermore, the game's unguided nature means that it is up to the individual player to find meaning within the game space, and one possible approach for the player to take is to go exploring the world. In this case a bigger world translates directly into more play hours that can be gained from this activity.

Of course in order to make exploration to be interesting for the player the world must present adequate variety to hold the player's attention. As an extreme example, simply using Perlin noise to generate a heightmap for the world and then filling it will create a world with infinite variety. However, in this case the player would notice the simplistic nature of the world generator and lose interest in

[1] A naïve implementation eventually runs into floating point precision errors, especially since world units in 3D graphics are usually implemented using the 32bit `float` type, rather than the 64bit `double` type (although using `double` would only push the problem out a little farther). This can be rectified by keeping the player character at the origin point and moving the world around them instead. Even if this is done, eventually numeric overflow will cause errors in the world generation algorithm and numerous other problems. However, in practice it would take so long to reach the world edge, as long as the player movement speed is restricted, that for all intents and purposes the world can be considered to have limitless size.

exploring the world almost immediately. So it is important then, to fill the world with a variety of interesting content. To this end such games will layer multiple noise functions to generate a multitude of terrain, and typically will create different biomes[2] (such as mountains, hills, tundra, desert, ocean, etc…). This process works well for natural terrain but is not suitable for generating "man-made" structures such as villages and castles. Yet fantasy architecture is a major feature in role-playing games such as World of Warcraft or Skyrim (an example of a castle in Skyrim can be seen in Figure 2). It adds character to the world and makes it feel inhabited, as well as providing incentive for the player to explore.

Figure 2 – Castle in Skyrim

Minecraft has made some limited attempts at integrating buildings in the world creation process and also now has small villages with multiple buildings. However, most of the buildings it creates are relatively simple. In order to ensure smooth placement, it will only spawn these in biomes that produce mostly flat terrain (for example villages only spawn in plains, savannah or desert biomes (Village - Minecraft Wiki)). Even so, the world generator sometimes runs into problems and produce broken results such as villages that are set two blocks higher than the roads they connect to.

[2] Biomes are major ecosystems that consist of a type of environment with vegetation and animal life adapted to it. It is term frequently used in voxel worlds to describe a distinct region created by the procedural terrain generator.

The aim of this project is to take a single kind of large structure, the castle, and devise a system for placing and generating it within a voxel world in a way that is plausible and well-adapted to the terrain.

This thesis begins with a review of materials that were considered during the design of the project in Chapter 2. We examine techniques used in medieval castle construction and the considerations of siege warfare. We then consider approaches to procedurally modelling architecture. After this some general techniques for working with voxels are looked at, such as sparse voxel octrees (a data structure designed for managing voxels) and techniques for smoothing surfaces generated from voxel data. Lastly the literature review will cover case studies for two of the biggest games using voxel technology so far: Minecraft and EverQuest Next, to see what approaches they have used for procedural generation of terrain and architecture, and how they have approached working with voxels in general.

In Chapter 3 the project design is introduced. It examines what was needed to make the project possible and divides it up into the major tasks that were undertaken. It also discusses the fact that the castle generation algorithm chosen is deterministic and that the random nature of the terrain generation process is the only source of variation used.

Chapter 4 is the bulk of this thesis and is where all the details of the implementation are covered. It first explains the basic voxel world engine that was created to make this project work. Then it discusses the terrain generation process employed in creating the terrains that the castles are to be built upon. It continues by explaining the drawing system employed by the project to make working with voxels easier. This is followed by a look at how the terrains were analyzed to allow searching for optimal placements for the castles. Lastly this chapter covers how castles were placed, had their layout optimized, and were constructed.

In Chapter 5 a look is taken at a number of demonstrations of the output generated by this project. We examine a number of terrains and look at the castles the algorithm generated for them. It includes subjective judgements as to how successful castle generation was for each of those cases. It also serves to demonstrate some of the variety that is possible for both the terrain generator and the castle generation algorithm.

This is followed by an evaluation of the project in Chapter 6. In this chapter we look at a user study that was done (in the form of an online survey) to gain feedback on how well the castle generation algorithm worked in the eyes of subjects that had knowledge of voxel world games like Minecraft but had no previous knowledge of this research. This chapter also includes a subjective evaluation of the results of the project by the author.

Finally Chapter 7 offers a conclusion to this thesis. A final look is taken at the work accomplished and how successful the project was. It also offers some suggestions for future work that could be done utilizing the research presented in this thesis.

Chapter 2: Literature Review

This chapter covers a number of topics relevant to this thesis. First we take a look at real medieval castle construction, looking at how siege warfare functioned and how this affected the decisions of real castle builders in times past (section 2.1 Medieval Castle Construction). Then we consider procedural architectural modelling methods to see how others have approached the overall problem of generating computer graphics models of buildings algorithmically (section 2.2 Procedural Architectural Modelling Methods). Once this basis is established a look is taken at general techniques for working with voxels that could be useful for the technical aspect of the project such as the sparse voxel octree data structure and methods for creating smooth surfaces from voxel data (sections 2.3 Sparse Voxel Octrees and 2.4 Voxel Smoothing Techniques).

The amount of formal literature available on voxel world games is sparse so far, but there is a lot of informal documentation on wiki-based sites, or as discussions on Internet forums. We will draw on this informal documentation to take a look at case studies of two different voxel world games: Minecraft and EverQuest Next. Minecraft is examined because it is the most well known voxel world game and because it was the inspiration for this project (section 2.5 Case Study: Minecraft). EverQuest Next is included because it shows that a voxel-based approach can be used to create a high-fidelity big budget title from a major video games developer (section 2.6 Case Study: EverQuest Next).

2.1 Medieval Castle Construction

As part of the work of designing the castle generating algorithm a look was taken at historical castle construction techniques with a specific focus on what defensive features would need to be considered.

During the siege of a castle the attacking force would surround the castle and cut off all routes of escape and supply (Stokstad, 2005). They would then construct a

camp fortified with palisades and ditches to strengthen their position. The next step would be to construct siege engines to try to break down the walls. A number of stone-throwing machines were also employed in siege warfare, such as the trebuchet or the mangonel. Another avenue of attack was the mining of tunnels under the walls when the castle was not built on solid rock or surrounded by water. The tunnels were propped up with timber while being excavated and then the timber supports would be set alight, collapsing the wall now bearing down on the tunnel with all its weight. Finally, there was the direct attack with knights, soldiers and archers. Soldiers would try to scale the walls with ropes or ladders, while archers provided covering fire.

While constructing their defences, castle designers would need to be aware of all of this. Castles were built on cliffs or were surrounded with natural defences or ditches, in order to make it harder to attack the walls with war machines such as battering rams or siege towers. For all of these avenues of attack it is advantageous to build the castle in a position with high ground advantage to be able to pick off attackers with archers on the wall, and to make it harder to reach with stone-throwing siege engines. It is also important that no nearby easily accessible points overlook the castle, or else they could be used by the attackers to fire upon the castle with the siege engines.

Medieval castles were frequently constructed to fit the local terrain (Gravett & Hook, 2007). For example see Figure 3 for the plans from a number of Welsh castles constructed in the late 13th and early 14th centuries. In these plans we can see that the castles are not constructed as regular shapes such as rectangles or octagons, but rather have been adapted to follow the features of the local terrain (the arrows around the castle plans show the slope of the terrain).

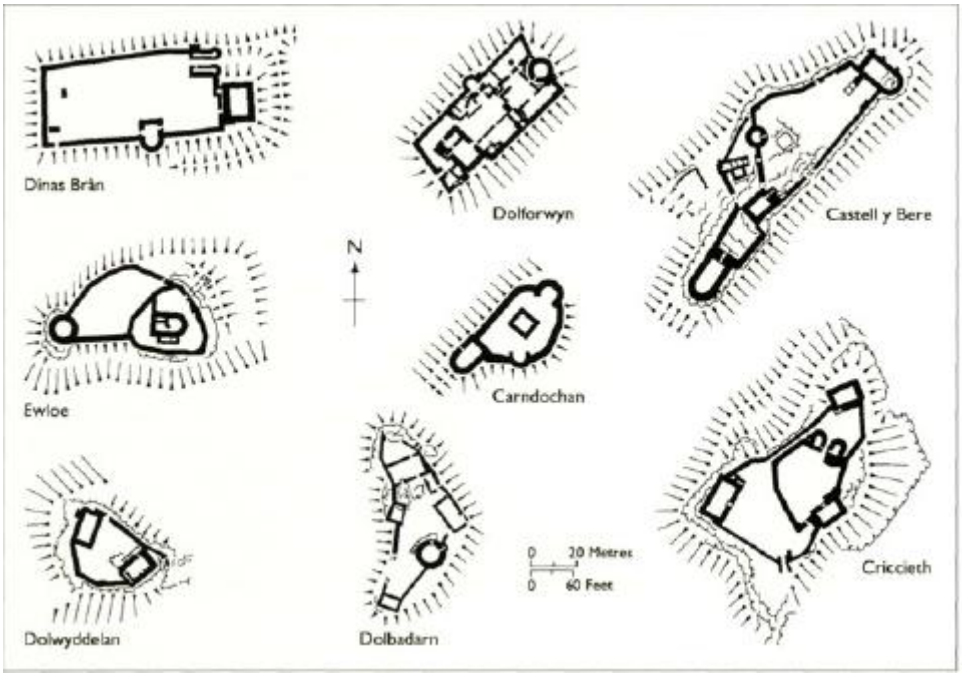


Figure 3 - Castles Built by Welsh Princes (Gravett & Hook, 2007)

2.2 Procedural Architectural Modelling Methods

In preparation for this project existing approaches to procedurally modelling architecture were examined. Typically these make use of some sort of production rule system such as L-systems (for an example of L-system output see Figure 4) or shape grammars. These define a number of production rules, which themselves can be comprised of further rules. They then include an initial rule that starts the production system. Rules may be applied serially or in parallel depending on the specific system implementation. In an L-system the output is then interpreted to produce a final result (for example a drawing or an architectural layout), while a shape grammar contains basic shapes that may be output directly to a working area as part of the production rules.

L-systems are well suited to creating shapes like trees that have a self similar nature. Consider the trees in Figure 4 that were created by the same L-system. A single rule (we can call it “branch”) is applied recursively and with a random component. The first time the rule is applied the trunk of the tree is created. The next recursion creates the main branches in random locations along the trunk (a scaling factor is also applied to shrink the size of the output). A further recursion

creates many small twigs along the branches. In this way a single simple rule can create a complex shape. By applying one or more random components to the rules (for example randomizing the placement, scaling and/or rotation each time the rule is applied) huge variation can be achieved in the output.

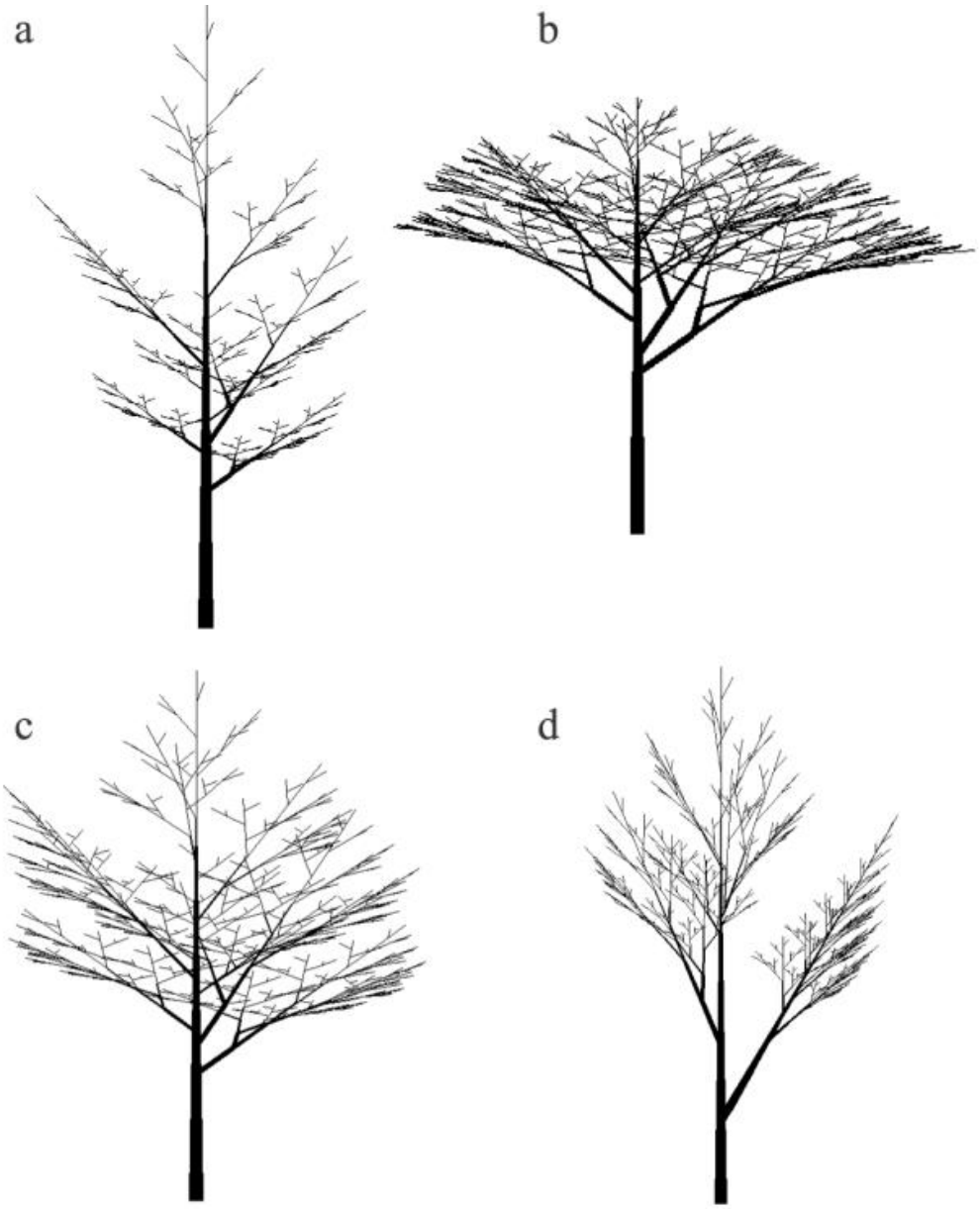


Figure 4 - Tree Shapes Generated by an L-System (Prusinkiewicz & Lindenmayer, 1990)

An example of this is the laying out architecture with the use of L-systems to create the blueprint of a city (Parish & Müller, 2001). They describe the use of what they all "self-sensitive" L-systems, capable of changing under local constraints to create a plausible layout for a city. The roads systems that make up

the underlying structure of a city are similar to trees, with major roadways being like trunks, streets are like branches and the side roads can be seen like twigs. Unlike when creating a simple tree care needs to be taken that streets do not overlap (or at least when they do intersections must be created) and we probably do not want two streets running parallel right next to each other (these sorts of considerations are solved by making the L-system "self-sensitive", i.e. changing on the local context that a rule is being executed in).

Similarly, the individual buildings themselves could also be constructed using a shape grammar (Müller, Wonka, Haegler, Ulmer, & Van Grool, 2006) (Hohmann, Krispel, Havemann, & Fellner, 2009). This approach has even been extended by making use of a node-based approach to creating the shape grammar, making it easier to use (Silva, Müller, Bidarra, & Coelho, 2013). This could eventually lead to this technology being used by game artists to create the massive amounts of content required by modern games. See Figure 5 for an example of buildings generated with the shape grammar approach.

Figure 5 - Example Buildings Generated with a Shape Grammar (Müller, Wonka, Haegler, Ulmer, & Van Grool, 2006)

Another approach is to combine stochastic optimization[3] together with a cost function to create optimal building layouts (Merrell, Schkufza, & Koltun, 2010). By combining machine learning with optimization techniques they manage to

[3] Stochastic optimization means that a large number of random layouts were tried and evaluated against a cost function. Better performing samples were kept and random variations were introduced. This process is repeated over many iterations, keeping layouts that perform well according to the cost function and discarding those that do not.

create a variety of visually plausible building layouts from a set of high level requirements. Due to the combinatorial explosion of different possible layouts an exhaustive search was deemed impossible in this case and for this reason a stochastic approach was used. Figure 6 shows a number of building layouts that were generated with this method.

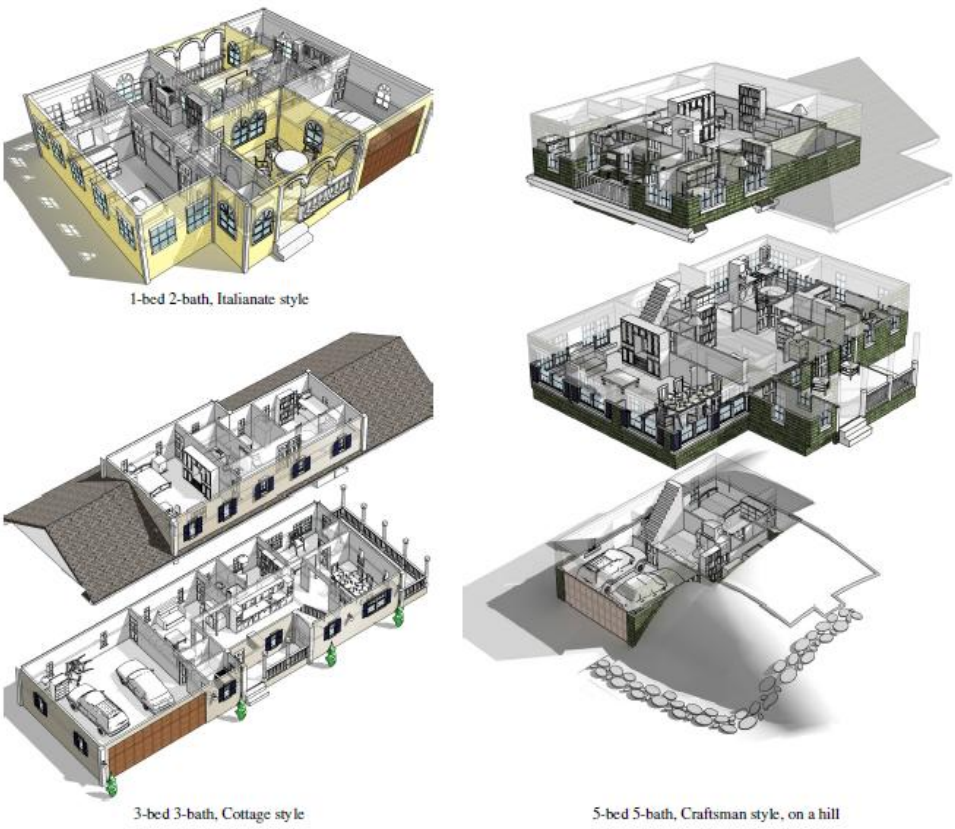


Figure 6 – Example Building Layouts Created with Stochastic Optimization (Merrell, Schkufza, & Koltun, 2010)

Both shape grammars and an optimization approach with a cost function were considered for this project. With a heavy focus on placing the castles and creating a layout adapted to the terrain the shape grammar approach was abandoned. Castle placement does make use of a cost function and optimization based approach, however in this case an exhaustive search is used (all of this is covered in detail in section 4.5.1 Placing the Castle).

2.3 Sparse Voxel Octrees

Moving on from the broad underlying concepts of castle construction and generating architecture procedurally we look at some of the technical details of working with voxels. When discussing voxels it is hard not to at least mention

sparse voxel octrees (SVOs). They are frequently mentioned in academic articles about voxels such as in (Laine & Karras, 2010). The SVO is a tree data structure in which every node is portioned into eight equal octants. The octree is a 3D version of the quadtree (an example visualization of a quadtree is shown in Figure 7). If at any point all leaf nodes would contain the same data (either they are all empty, or they all have the same voxel type) the tree terminates at that point (hence the "sparse" in sparse voxel octree). This is a highly efficient way to store voxel data, especially in cases where many large volumes are empty.

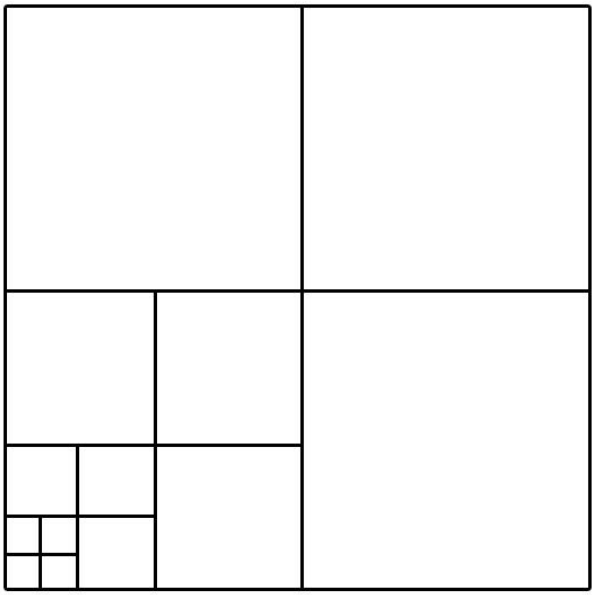


Figure 7 - A Visual Representation of a Quadtree

The SVO is also a data structure well suited to ray casting into, something that has been exploited to create very fast methods for directly rendering voxels without need to turn them into polygons first (Crassin, Neyret, Lefebvre, & Eisemann, GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering, 2009).

On the other hand SVOs have some disadvantages for voxel world games as well. They are more complex to implement than a simple array-based approach and adding, deleting and modifying blocks is potentially an expensive operation (potentially causing a large part of the octree needing to be rebuilt). Some voxel world games utilize SVOs to store their data (for example EverQuest Next) while others, such as Minecraft, do not.

2.4 Voxel Smoothing Techniques

Voxels generally need to be converted to a polygon mesh for display (although direct ray casting methods do exist, as mentioned in section 2.3 Sparse Voxel Octrees). During the process of converting voxels from their internal representation as a uniform grid of values to polygons (i.e. creating an isosurface) for display there are several options. One route is to convert to a blocky box-based representation, such as that seen in Minecraft and many similar voxel world games (Figure 8).

Figure 8 - Minecraft

If we want a smoother looking polygon mesh we need to do some extra work (this is analogous to the concept of anti-aliasing in 2D raster graphics).

One option is to make use of the Marching Cubes algorithm. Originally developed for use with computed tomography (CT), magnetic resonance (MR) and single-photon emission computed tomography (SPECT) data in order to create smooth models; marching cubes is a relatively simple algorithm that uses a pre-computed table of shapes to create mesh data (Lorensen & Cline, 1987). A logical cube is moved along the voxel or point cloud data and all eight neighbours (the vertices of the logical cube) are then considered. If they are considered solid for the purposes of the surface we are constructing then that vertex in the logical cube is set to one, otherwise to zero (it is important to note that the algorithm was initially designed for density data where a threshold value would be used to define what is

considered inside or outside of the surface – however in a voxel world game the voxels are clearly defined as being either solid and inside the surface, or not). There are eight vertices in the cube and two possible states for each (inside the surface and set to one, or outside the surface and set to zero) for a total of 256 combinations. Due to the symmetries that exist, this can be narrowed down to just 15 unique cases, displayed in Figure 9 below:

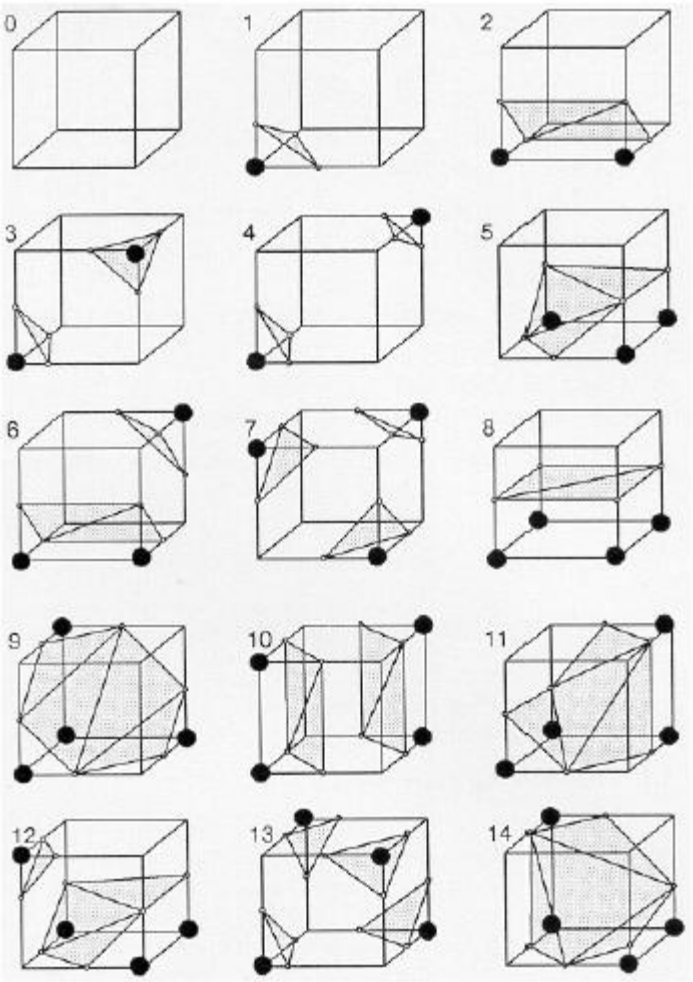


Figure 9 – Marching Cubes Unique Cases
(Lorensen & Cline, 1987)

Creating a polygon surface then is as simple as calculating the values (1 or 0) for each of the vertices in the cube and then making a lookup into a table that contains all of the possible cases and the mesh data that they generate. This method can generate some ambiguous cases, an issue that can be resolved by making use of the asymptotic decider algorithm developed by Nielson and Hamann (Nielson & Hamann, 1991). This adds considerable complexity to the algorithm but does resolve the ambiguous cases that can arise in the standard marching cubes

algorithm. Figure 10 shows the Stanford Dragon rendered with a cube representation on the left and using the marching cubes algorithm on the right, showing the smoothing this creates.

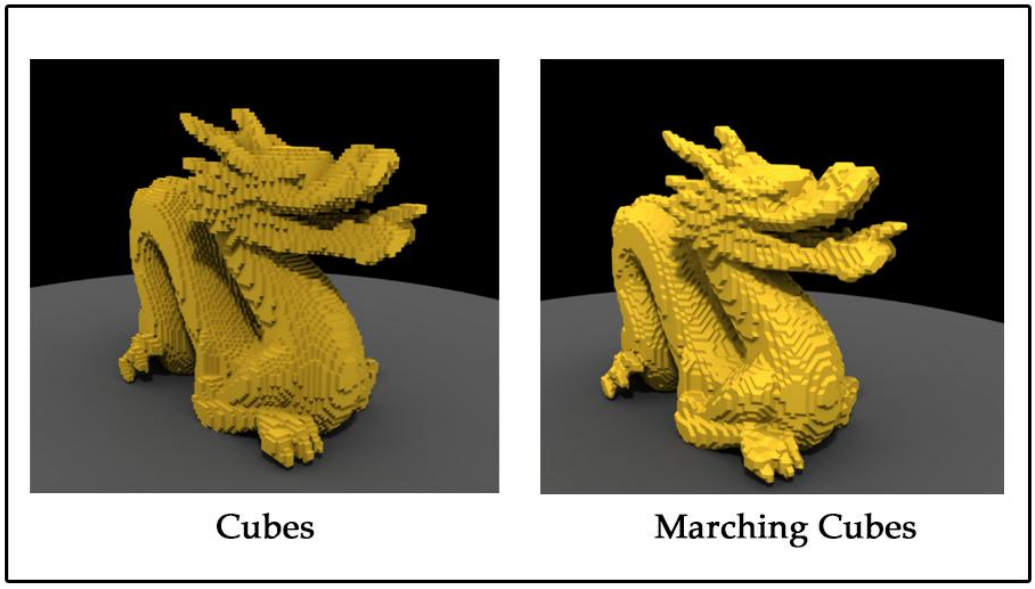


Figure 10 - Stanford Dragon Cubes to Marching Cubes Comparison (Image Generated in MagicaVoxel)

A further option for creating smooth surfaces from voxel data is dual contouring. This is an extension of marching cubes and surface nets (yet another isosurface generation algorithm). By making use of a grid that is tagged with Hermite data (the intersection points and normals) this method is capable of generating an isosurface that contains both smooth and sharp edges (Ju, Losasso, Schaefer, & Warren, 2002). Figure 11shows a mechanical part rendered with dual contouring. Note how it contains both hard and smooth edges. Dual contouring also operates on an octree representation of the voxel data internally, making it capable of generating a surface mesh at different resolutions, perfect for creating different levels of detail of an object so that lower detail levels can be shown for objects far from the player in a game environment.

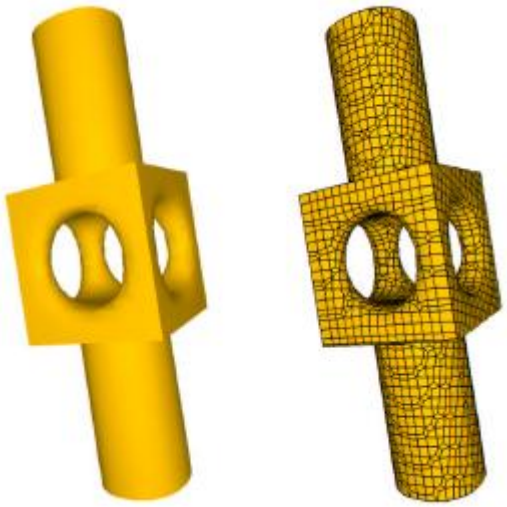


Figure 11 - Mechanical Part Rendered with Dual Contouring (Ju, Losasso, Schaefer, & Warren, 2002)

2.5 Case Study: Minecraft

Easily the most well known of the voxel world game is Minecraft. First released in 2009 and acquired for USD$2.5 billion by Microsoft in 2014 (Peckham, 2014), Minecraft is still by far the most popular of the voxel world games that have been released so far. In Minecraft players are largely left to their own devices, with gameplay driven by the creative choices of the players themselves, rather than the rule-set and goals prescribed by the game. Especially the creative mode in the game (where players have access to unlimited blocks of all types, are invulnerable, and can fly at will) resembles a simple CAD program more than a true game.

Players can place many different types of blocks into the world to build their creations. These include regular cube shapes with different types of textures applied but also a variety of more complex shapes such as torches, fences, and doors. By making creative use of these more complex shapes players are able to create designs with more apparent geometric complexity than a $1m^3$ block size would otherwise suggest.

Figure 12 - Sample of Block Types from Minecraft

As of the 1.8 Beta edition of Minecraft released on the 15th of September 2011 Minecraft is able to generate villages in the world. These are constructed procedurally out of prebuilt parts. They are only built in flat areas of the world in order to make the placement simple. An example can be seen in Figure 13. Note how all the buildings are generated on the same plane on the ground.

Figure 13 – Minecraft Village
(Village - Minecraft Wiki)

In the same 1.8 Beta update there was the addition of "strongholds". These are complex structures that spawn underground. Because they are generated entirely underground they do not have to take into account the surrounding terrain, they are simply generated and displace the voxels that otherwise would occupy the space. In Figure 14 we can see a view of a stronghold completely uncovered. Note

that none of the inside of the stronghold is visible since it generates walls that encase it completely. This creates a complex structure but makes no use of the terrain it is in, other than making sure that it is entirely underground.

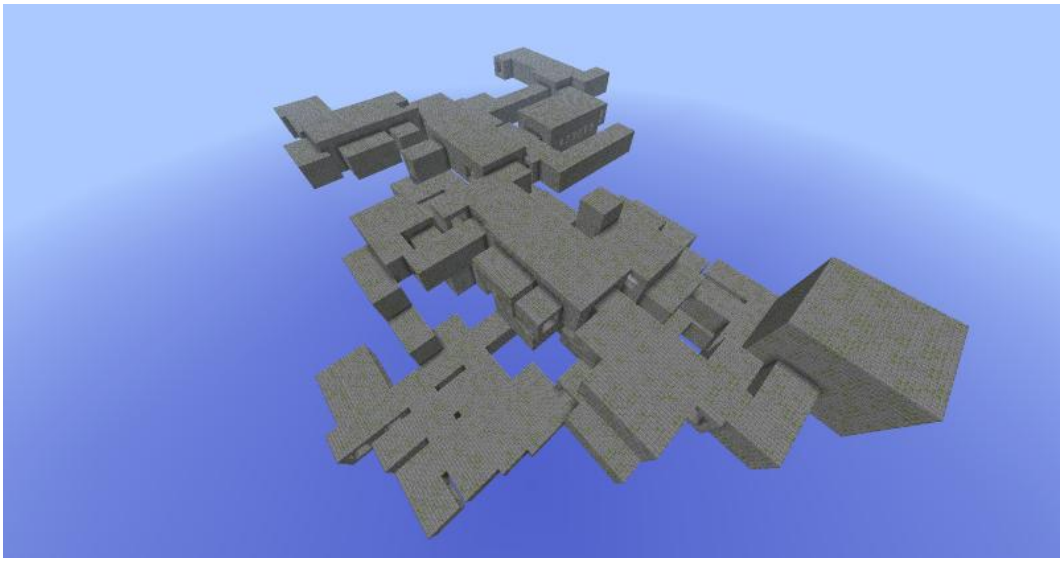


Figure 14 - Minecraft Stronghold Uncovered
(Stronghold - Minecraft Wiki)

Players have created a number of impressive castles. These were crafted by hand, rather than procedurally generated but in Figure 15 and Figure 16 it can be seen what is possible within the constraints of working with a relatively low resolution world such as the one in Minecraft. The images show that it is possible to create impressive castle structures within the constraints of Minecraft.

Figure 15 – Minecraft Castle on a Mountain
(Bear, 2013)

Figure 16 – Minecraft Castle Ruin
(Bear, 2013)

2.6 Case Study: EverQuest Next

Another game to make use of voxels is the upcoming next instalment in the popular massive multiplayer online (MMO) game franchise EverQuest. It is notable because it is the first AAA (big budget) game title to make use of voxels as a major component in recent years (CryEngine, the engine used in the first person shooter franchise Crysis, included some support for voxel based terrain in the editor although this has been deprecated as of version 3.5 (Crytek, 2014)). It also uses the underlying voxel technology to produce worlds with high graphical fidelity unlike Minecraft and its clones. An example screenshot from the game can be seen in (Figure 17). All elements in that image, except for the player character and the sky, are generated from an underlying voxel representation. The voxel size used is smaller than the typical $1m^3$ used by Minecraft, being about $0.2m^3$ per voxel (Voxel - Landmark Wikia, 2014). This smaller voxel size, combined with the fact that crease data is stored per voxel, means that scenes can have graphical complexity approaching that of games using a purely polygon-based approach.

Figure 17 - EverQuest Next
(Haas, 2013)

For the voxel elements of the engine EverQuest2 makes use of the commercially available VoxelFarm engine that grew out of a series of voxel terrain experiments by Miguel Cepero (Voxel Farm, 2015) (Cepero, Procedural World: EverQuest Next, 2013) (Khaw, 2013). To create the isosurface from the voxel representation VoxelFarm uses the dual contouring method (Cepero, Procedural World: From Voxels to Polygons, 2010). As mentioned in section 2.4 Voxel Smoothing Techniques, this means that the world can be created with both smooth and hard edges.

In an interview Steve Klug, the technical director of EverQuest Next, said this about voxels: “Creating a world that allowed the dynamic interaction we were envisioning drove the decision. We looked at a number of different approaches and the voxel solution was the most appropriate” (EQNexus, 2013). Further on in the interview he continues to talk about choosing voxels specifically to allow for a user editable world. One in which things could be destroyed by the players, but also one that the players could build in. This ability to edit worlds once deployed to players is one of the key features that make voxels a compelling technology for future games. Importantly, the tools used to edit the world are simple to use, yet powerful. Per the interview with Steve Klug quoted above, the tools used by players in the game will be exactly the same as those used by the developers.

EverQuest Next shows a different usage of the voxel world paradigm, with higher fidelity graphics and a mixture of handcrafted [4] and procedurally generated elements. It demonstrates that a voxel representation approach is of interest to future AAA games development, and shows that the work done in this project may be applicable to a wide variety of games in the future.

Although architecture in EverQuest Next will likely be handcrafted, the creator of the underlying voxel engine (the VoxelFarm engine), has used shape grammars to construct elaborate architecture, including castles (although placement into the world was done manually) (Cepero, Procedural World: Castle by the lake, 2015). Figure 18 shows what such a castle looks like and how much detail is possible. Although the castles created by this project do not approach that level of graphical fidelity, they are created entirely without human input (unlike those in VoxelFarm that are manually placed).

Figure 18 - Castle in VoxelFarm
(Cepero, Procedural World: Castle by the lake, 2015)

2.7 Summary

This chapter has outlined a number of topics relevant to this thesis. It discussed medieval castle construction and how builders of these castles would need to take into account the realities of siege warfare. This provides a basis making decisions

[4] The EverQuest Next team has released a sandbox game called Landmark for the express purpose of allowing players to create buildings and terrain features that may be included in the final game (Daybreak Game Company LLC, 2015).

on how to place and construct castles (section 4.5 Castles). We then looked at procedural modelling techniques for creating architecture. This looked at L-systems and shape grammars, which were considered but not used for this project, and a technique using a recursive optimization function that inspired the castle layout optimization process used (explained in section 4.5.2.3 Optimization Step).

In addition we examined techniques for working with voxels, including the marching cubes algorithm for creating smooth surfaces from voxel data. A simplified version of this approach was used to create a smoother look for the terrain (see section 4.2.10 Smoothing the Terrain).

Lastly we considered two examples of voxel world games. Looking at Minecraft showed us some attempts at introducing procedurally generated architecture to voxel world games in the form of villages and strongholds. It also showed us that a voxel cell can be converted to more complex shapes than just cubes in mesh creation (for example stairs or torches), and this was adapted for this project in the form of MicroBlocks (details in section 4.1.3.4 MicroBlocks). The example of EverQuest Next showed that high-fidelity graphics can be achieved with a voxel approach and shows that this research could be applicable to future big-budget titles. Even though it uses procedurally generated terrain it still makes use of manual content creation for buildings, suggesting that procedurally generating architecture in voxel worlds is an open problem.

Chapter 3: Project Design

The common problem among voxel world games is providing the player with an interesting environment to explore. One element in particular that is either missing or only poorly implemented is any form of man-made structure. In order to look plausible such structures must appear to have been placed and constructed by an intelligent agent. Since the world generation algorithms used make use of pseudorandom processes in the form of coherent noise functions they are poorly suited to producing features with such an appearance of agency. Simply placing a structure at a random point in the world will frequently result in poor or even disastrous results such as placing a castle right into a steep mountain or on the ocean floor. In the literature review section on Minecraft (see 2.5 Case Study: Minecraft), we saw that restricting placement of structures to flat terrain is a possibility but leaves us with limited flexibility. Instead, this project set out to place a castle into a voxel world and make use of the procedurally generated terrain. The aim was to avoid problematic areas where a castle could not be built (such as into steep mountains) while favouring high ground and adapting its layout to the terrain.

In order to produce a sensible result the project then required to be divided into four steps: the underlying voxel engine (see section 4.1 Voxel World Basics), the initial terrain generation (see section 4.2 Terrain Generation), then the castle placement (see section 4.4 Terrain Analyzer and section 4.5 Castles) and lastly the actual castle construction (covered in section 4.5.3 Constructing the Castle).

The underlying voxel engine was developed on top of the Unity3D engine using Minecraft as inspiration for many implementation details such as using chunks (see section 4.1.3 Chunks) and how lighting is done (see section 4.1.3.1 Lighting for details).

Terrain generation uses a number of pseudorandom noise functions combined together to create a heightmap and to populate the world with blocks (discussed in section 4.2 Terrain Generation).

For the approach to the castle placement and construction a shape grammar based approach was initially considered (see section 2.2 Procedural Architectural Modelling Methods). However, shape grammars are generally ill suited to taking outside factors into account such as the shape of the terrain. For this reason it was decided to use a search-and-optimize approach to the castle placement and layout generation (as decided in section 2.7 Summary). It would still be possible to make use of shape grammars to create individual elements of the castle such as towers and buildings but this was not done for this project in the interest of concentrating on the placement. This meant that the castle construction was done in a completely deterministic manner. Variation would be achieved by adapting to the randomly generated terrain, and not by adding random elements to the castle generation step.

The process of developing the actual castle placement and layout creation algorithm (covered in section 4.5 Castles) focused on generating what would be considered a plausible result by players of a voxel world game (a survey of such players was conducted to evaluate the success of this and is discussed in section 6.1 User Study). A number of test terrains were generated with castles and problems were identified. Then heuristics were added to improve the results as much as possible. The issues specifically targeted were: avoid problematic areas such as steep mountains and gorges/canyons where possible while still making good use of the terrain overall (we do not want to restrict castle placement only to flat areas but nor do we wish to place the castle into the side of a steep mountain or spanning a canyon). Secondly, try to place the castle in a good defensive position. For the purposes of this project this meant optimizing for the most fundamental of military advantages: occupying the high ground.

Castle construction was mainly done with the aid of a drawing system adapted from 2D (working with pixels) to 3D (working with voxels) for this project (see section 4.3 Drawing with Voxels). Once the castle placement and layout is done this is a relatively simple process (see section 4.5.3 Constructing the Castle).

Chapter 4: Implementation

An initial prototype of this project was attempted using C# and the SlimDX API port of Microsoft™ Direct3D11 (see Figure 19 below). Although this progressed to a very basic functional stage, it quickly became apparent that as much time would be spent on the technical details of essentially creating a game engine from scratch as would be spent on the project proper (every additional feature would require more coding work to implement from scratch, such as adding a skybox, shadows, transparency, collision detection and more). For this reason it was decided to switch to using the free version of the Unity3D game engine instead.

Figure 19 – Early Prototype

The Unity3D engine (www.unity3d.com) provides all the features required for this project. In particular it makes it easy to procedurally generate meshes, and provides built-in camera controllers as well as collision detection. The only feature missing from Unity3D that would have been useful for this project is a Direct3D10+ feature called texture arrays. This was used in the initial prototype but then replaced with a texture atlas in the Unity3D version (more on this in section 4.1.2 Blocks / BlockEditor).

The "Personal Edition" is free to use (for projects making less than US$100,000 per year), and as of version 5 includes all engine features (previously advanced

features such as deferred shading and image effects were reserved for the Pro version). Unity3D features three options as scripting languages to implement the game logic: C#, JavaScript and Boo. This project uses the C# option exclusively, because it is the recommended language (Aleksandr, 2014) and because I personally have the most experience with it over the other options. It is also the only way to have access to the full .NET 2.0 API libraries, including in particular the generic collection classes such as Dictionary<T> and List<T>.

The first step to realizing this project was to create a useful voxel world engine on top of Unity3D (covered in section 4.1 Voxel World Basics). This had to be functional as well as performing fast enough to be practical. Even with the massive performance of today's computers voxel worlds can still quickly tax them beyond their limits if great care is not taken during implementation. This is because even a relatively small area may contain many tens of millions of voxels (with a voxel cell size of $1m^3$ a volume just $100m^3$ in size contains 1,000,000 voxels).

In addition to a voxel engine capable of managing a reasonable sized voxel world and converting it to polygon meshes for display, a prerequisite to this project was a terrain generator capable of creating varied terrain in a style similar to voxel world games like Minecraft (this is discussed in section 4.2 Terrain Generation).

After terrain was done we needed a method for working within the voxel world to examine and place blocks efficiently. My approach to this was to adapt 2D drawing functions designed for working with pixels to 3D to use voxels instead (explained in section 4.3 Drawing with Voxels).

Before we could place the castle into the terrain we would need to analyze it so that a good spot could be chosen. How the terrain was analyzed is explained in section 4.4 Terrain Analyzer.

With all of these building blocks in place it was then possible to search for a good place for a castle, optimize the castle layout and finally construct the castle in the voxel world (discussed in section 4.5 Castles).

4.1 Voxel World Basics

This part of the thesis explains in detail how the voxel world engine was constructed within Unity3D.

Voxel world engines tend to share a number of elements. At the core they have a number of different types of blocks, which can be placed into the world in voxel cells (elements in a regular 3 dimensional grid array). The voxel storage array itself is usually broken up into larger elements called chunks, although other storage methods such as sparse voxel octrees (discussed in section 2.3 Sparse Voxel Octrees in the literature review) are also possibilities.

Blocks stored within the voxel grid are then converted to polygons by some means for display. Polygon conversion may be as simple as creating a box shape the size of one voxel cell (this is true for most block types in Minecraft for instance), creating a more complex mesh in the voxel cell location (such as a staircase or torch in Minecraft), or may use a more complex algorithm like marching cubes or dual contouring to create a smooth surface (see section 2.5 on Voxel Smoothing Techniques in the literature review).

This project also uses a layer system similar to that found in many 2D drawing packages, adapted to voxels. The motivation for this was the ability to quickly toggle between showing just the terrain, and showing the terrain with the castle constructed in it (how this works is discussed in detail in section 4.1.3.5 Layers).

In order to populate the world with blocks for the player a procedural algorithm is used. While the specifics of world generation differ between voxel world games, they commonly use some sort of coherent noise function (see section 4.2.1 Libnoise later in this document for more on coherent noise) as the basis.

The following section explains how I tackled the challenges of building a voxel world engine and creating a base terrain in which to construct a castle.

4.1.1 Introducing VoxBox

For the purposes of this project I created a voxel sandbox that I call VoxBox (short for Voxel Sandbox). This section discusses the underlying architecture of VoxBox.

For world-scale I chose to stick with the Unity3D recommended (and intuitive) size of 1.0 world unit being equal to 1 metre. In terms of voxel grid scale, I decided to use the same scale as Minecraft: 1 voxel cell is $1m^3$ in size. This was done for two reasons: firstly it is a common choice in voxel world games (used in Minecraft, FortressCraft, Minetest, among others) and thus output from the project would be immediately able to be compared to these popular existing games. The second reason is simply that it is a size that strikes good balance between world fidelity and performance. Halving the voxel cell size to $0.5m^3$ means an 8x increase in the number of voxels required to fill a given area, meaning a significant increase in memory usage, CPU processing of lighting, and GPU usage to render the resultant meshes (all roughly linear with the increase in the number of voxels). A much faster way to increase world fidelity as needed is discussed in section 4.1.3.4 - MicroBlocks.

In order to keep the user interface responsive while the system is processing the world (for example when it is generating the terrain during initialization) two threads were used. A foreground thread for tasks that must be done on the main thread (any calls to Unity3D API functions must be done from the foreground thread) and a background thread for the heavy processing (such as terrain generation, lighting calculations, castle generation, and most of the mesh generation). Tasks can be scheduled on either the foreground or the background thread as needed. Task scheduling uses a simple queue (FIFO) buffer.

Note on the coordinate system: the 3D coordinate system used by VoxBox is the same as the default Unity3D coordinate system. That is the y-axis goes up and down, and the horizontal plane is made up of the x and z-axis. Some engines such as UnrealEngine 4 use the z-axis as the vertical axis and have the x and y-axis describe the horizontal plane. This is a minor detail and the choice is just a matter of preference, but it is important to know this for when some of the implementation details are discussed later in this chapter.

4.1.2 Blocks / BlockEditor

For the blocks the Flyweight pattern[5] (Nystrom, 2014) was used. Each block is stored as a pointer to an instance of the Block class that contains all the information about the block. A block manager stores all the available block instances (one for each type of block available for use in this voxel world, such as dirt or stone) and allows looking them up by name or an ID number.

Since blocks will have varying textures depending on type, a texture atlas is created when the program starts. The initial prototype program used an advanced Direct3D API feature called texture arrays for this, which as the name implies allows you to specify an array of textures to use with a mesh object where an index to the array is included with each vertex. Since Unity3D does not support texture arrays (at least not as of version 5.1, the version used for this project), a different solution was needed. The options were to either create a separate mesh for every texture used, exploding the number of draw calls that would need to be made, or pack all block textures into a texture atlas. That is, all of the textures used are placed into a single texture. This allows the same texture to be used across the entire terrain mesh, allowing it to be rendered as a single draw call. Using a texture atlas therefore offers large performance benefits and this route was chosen. Next there were two options for creating the texture atlas: manual creation using a program like Photoshop™ or automatic creation by VoxBox. Manual creation would have been slightly quicker to get running initially but would have meant more work when adding new textures later, so for this reason the second route (automatic generation) was chosen. Luckily Unity3D has a built in function, `Texture2D.PackTextures()`, that makes this relatively easy. See Figure 20 to see what the texture atlas created by VoxBox looks like.

[5] The flyweight pattern is a software design pattern that minimizes memory usage by sharing as much data as possible between similar objects. By using references back to a shared instance of an object redundant data storage in memory is minimized.

Figure 20 - Texture Atlas Created by VoxBox

The Block class is simply a collection of information about each type of block in VoxBox, and has the following properties:

- `Name` (`string`)
- `ID` (`int`)
- `IsNatural` (`bool`)
- `IsSolid` (`bool`)
- `IsTransparent` (`bool`)
- `BlocksLight` (`bool`)
- `IsSmooth` (`bool`)
- `Geometry` (`IGeometryBuilder`)
- `TopFace` (`UVRect`)
- `BottomFace` (`UVRect`)
- `LeftFace` (`UVRect`)
- `RightFace` (`UVRect`)
- `FrontFace` (`UVRect`)
- `BackFace` (`UVRect`)
- `Light` (`Color32`)
- `Cost` (`int`)

`Name` and `ID` both provide ways of identifying blocks. Further on in this document when needing to refer to a specific type of block, the name of the block is written in quotes. For example: “Air”, “Grass”, or “Dirt”.

`IsNatural` was going to be used to distinguish blocks generated by the terrain generator and those placed by the castle builder, but this was abandoned and partly replaced by the layer system (see 4.1.3.5 Layers).

`IsSolid`, `IsTransparent` and `BlocksLight` all appear to have similar function but there are some important differences. `IsSolid` specifies if the block should be included in the collision mesh construction. If set to true, the block is added to the collision mesh and as such impassable to the player. `IsTransparent` is used during display mesh generation (see section 4.1.3.2 Conversion to Polygons). If this is set to true it means that any neighbours will generate a face where they share a side. `BlocksLight` is used during the lighting calculation (see section 4.1.3.1 Lighting), and naturally if set to true it prevents light from spreading across this block. So for example “Glass” is solid (blocks player movement), transparent (neighbouring blocks must have faces touching this block created), and does not block light. On the other hand “OakLeaves”, the block used for leaves on the trees is solid and transparent, but does block light (creating somewhat shadowed areas under trees).

`IsSmooth` is used during mesh creation to smooth the normals of blocks marked with this property. By default normals are calculated per side of each block, if this property is set the vertices that share a single point in space for this block are averaged out to give a more smooth appearance. Figure 21 shows a cube with flat normals to the left and a cube with smooth normals on the right. Vertex normals are shown as short lines at each of the cube corners. The `IsSmooth` property is only used for the grass blocks in this program.

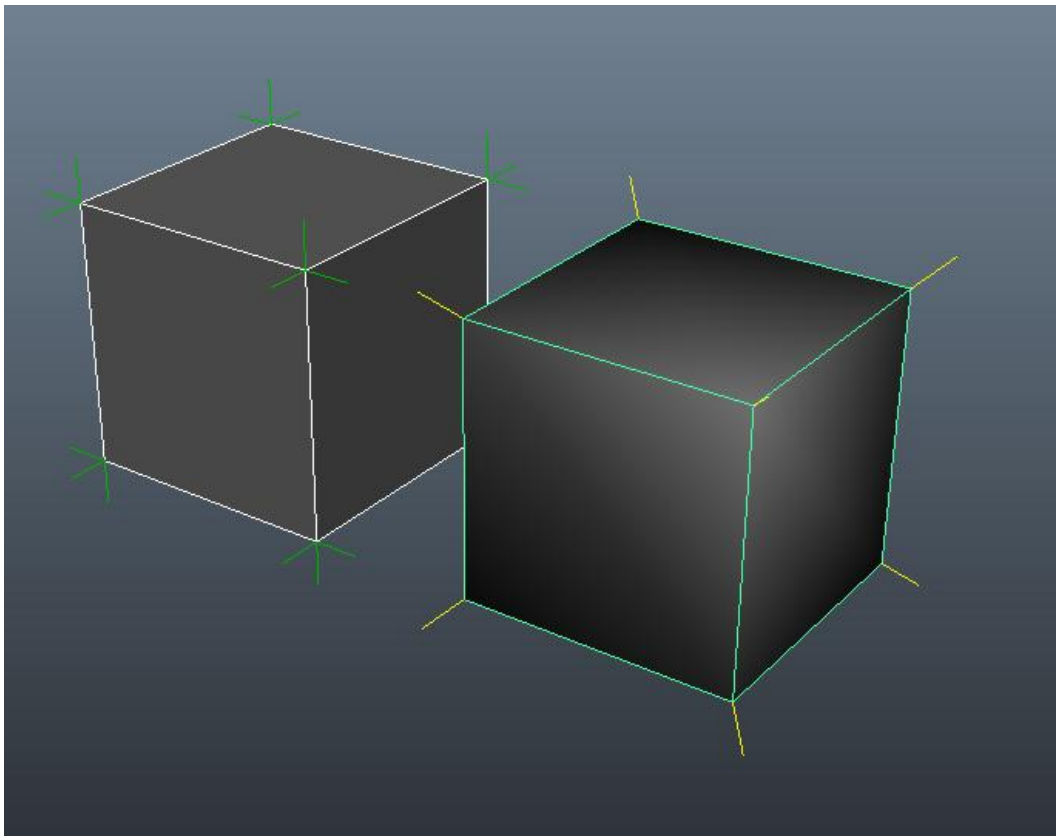


Figure 21 - Flat vs Smooth Normals

`Geometry` is a pointer to an instance of a class implementing the `IGeometryBuilder` interface (an interface created for this project, for more information see section 4.1.3.3 later on in this chapter). The interface has a single function prototype: `BuildGeometry()`. This is called for each non-"Air" (i.e. not empty) block during mesh construction. Simple blocks use the `BoxBuilder` implementation of this interface, creating a solid cube in the space for this block. Geometry builders are discussed in more detail later on in 4.1.3.3.

The six `UVRect` values (`TopFace`, `BottomFace`, etc…) encode the position on the texture atlas to use for each side of the block. For most blocks these values are identical but for example "Grass" blocks have a grass texture on top, dirt on the bottom and a grass-to-dirt transition on the sides. Figure 22 shows a grass block on a dirt block with the polygon mesh visualized. As can be seen the top face is completely green while the side faces show a transition from the green grass to the brown dirt.

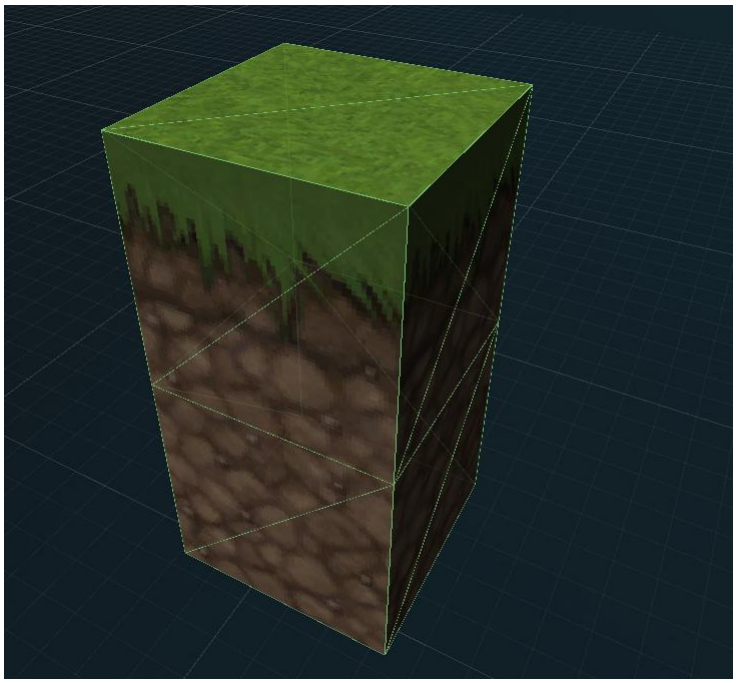
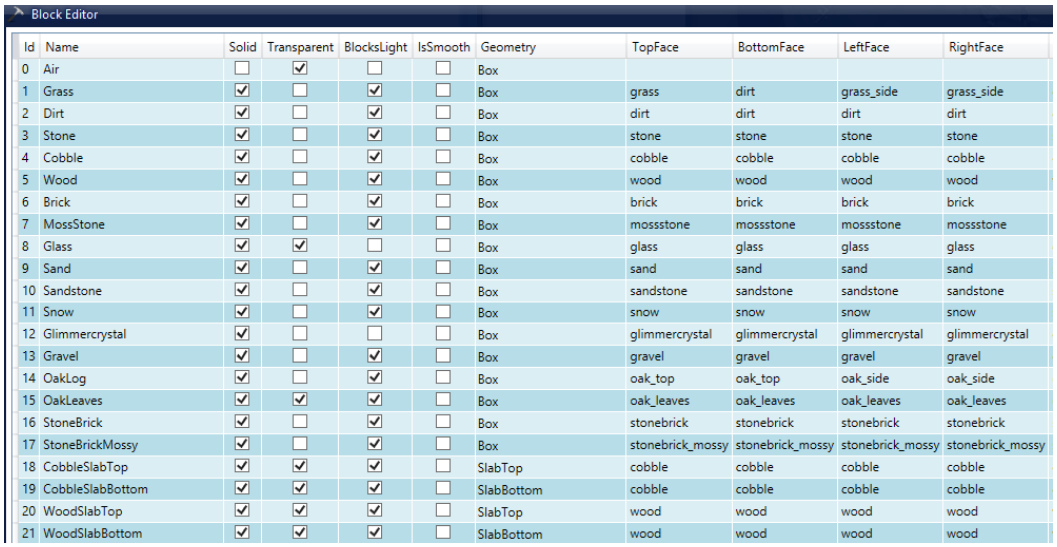


Figure 22 - A Grass Block on a Dirt Block

The `Light` value is used for blocks that emit their own light. This could be used for torches for example. While no blocks in this project emit large amounts of light, they all have a low, but non-zero, value set for this to add some ambient lighting to the world (see section 4.1.3.1 Lighting for more information on how lighting is computed).

The `Cost` can be used to calculate the cost of building a given castle. This is arbitrary and would depend on a specific game application. A total castle cost value is calculated but not used in the optimization process. It was considered but was found infeasible, because to calculate the cost the entire castle must be built (i.e. all the voxels for the castle must be placed in the world), a process that takes several seconds. The castle placement algorithm tries many possible castles, and building each of them fully to get the cost value would make this process potentially take hours (rather than constructing the castle at each point, the optimization algorithm instead simply checks the computed tower and wall positions against the terrain – for more details see section 4.2.5 on castle placement). However, for example, in a strategy game making use of the castle generation algorithm this might be used to present the player with a cost value for constructing the castle.

In order to simplify the process of adding new types of blocks to VoxBox a block editor program was developed in C# and using Windows Presentation Foundation. It consists of a single form with a datagrid component for adding and modifying block types. This is saved to an XML file that is then read in by VoxBox on start up to initialize the block manager with all the available block types.



Id	Name	Solid	Transparent	BlocksLight	IsSmooth	Geometry	TopFace	BottomFace	LeftFace	RightFace
0	Air	☐	☑	☐	☐	Box				
1	Grass	☑	☐	☑	☐	Box	grass	dirt	grass_side	grass_side
2	Dirt	☑	☐	☑	☐	Box	dirt	dirt	dirt	dirt
3	Stone	☑	☐	☑	☐	Box	stone	stone	stone	stone
4	Cobble	☑	☐	☑	☐	Box	cobble	cobble	cobble	cobble
5	Wood	☑	☐	☑	☐	Box	wood	wood	wood	wood
6	Brick	☑	☐	☑	☐	Box	brick	brick	brick	brick
7	MossStone	☑	☐	☑	☐	Box	mossstone	mossstone	mossstone	mossstone
8	Glass	☑	☑	☐	☐	Box	glass	glass	glass	glass
9	Sand	☑	☐	☑	☐	Box	sand	sand	sand	sand
10	Sandstone	☑	☐	☑	☐	Box	sandstone	sandstone	sandstone	sandstone
11	Snow	☑	☐	☑	☐	Box	snow	snow	snow	snow
12	Glimmercrystal	☑	☐	☐	☐	Box	glimmercrystal	glimmercrystal	glimmercrystal	glimmercrystal
13	Gravel	☑	☐	☑	☐	Box	gravel	gravel	gravel	gravel
14	OakLog	☑	☐	☑	☐	Box	oak_top	oak_top	oak_side	oak_side
15	OakLeaves	☑	☑	☑	☐	Box	oak_leaves	oak_leaves	oak_leaves	oak_leaves
16	StoneBrick	☑	☐	☑	☐	Box	stonebrick	stonebrick	stonebrick	stonebrick
17	StoneBrickMossy	☑	☐	☑	☐	Box	stonebrick_mossy	stonebrick_mossy	stonebrick_mossy	stonebrick_mossy
18	CobbleSlabTop	☑	☑	☑	☐	SlabTop	cobble	cobble	cobble	cobble
19	CobbleSlabBottom	☑	☑	☑	☐	SlabBottom	cobble	cobble	cobble	cobble
20	WoodSlabTop	☑	☑	☑	☐	SlabTop	wood	wood	wood	wood
21	WoodSlabBottom	☑	☑	☑	☐	SlabBottom	wood	wood	wood	wood

Figure 23 – BlocksEditor

4.1.3 Chunks

Voxel world games typically divide the world up into units called “chunks”. The size of chunks varies between implementations, although sizes of 16^3 or 32^3 are common. Minecraft uses a slight modification on this system. It uses a fixed world height of 256 blocks, with one chunk being 16×256×16 blocks. Each of these is further divided into 16 units of 16^3 blocks for rendering. To maintain more flexibility vis-à-vis world height I decided not to use the Minecraft model and stick with simple 16^3 blocks sizes for chunks in VoxBox so that as many chunks as needed could stacked vertically.

There are several reasons for dividing up the world in chunks, rather than treating it as a single monolithic entity. The first is that it allows parts of the world to be loaded/unloaded dynamically as the player moves around, which allows for worlds far larger than can be drawn at once. For this project this wasn’t a concern as only a small area is created. Another benefit is that when changes are made to the world only affected chunks need to be updated (such as when the player modifies a block or in the case of this project when a layer is toggled on or off).

Since recreating the lighting and mesh data is an expensive operation this is highly desirable. However, the biggest benefit is in rendering. By splitting up the terrain mesh into chunks, any of those not visible to the user (i.e. not in the camera view frustum) can be culled prior to rendering, saving GPU power and increasing framerate. As mentioned previously this also allows us to stack chunks as far as needed vertically. This comes in useful when creating large mountains (for more on mountains and the terrain generation process see 4.2.6).

Besides these concerns, the Unity3D Mesh primitive uses 16bit index buffers, putting a hard limit of 65,536 vertices on any single mesh. Using a chunk size of 16^3 gives us 4,096 blocks per chunk. The worst case scenario (assuming simple cube geometry) is if every second block is set (imagine a 3D checkerboard pattern), giving 2,048 cubes, each with 6 sides. Each side uses 4 vertices (two corner vertices are shared, faces cannot share corner vertices with each other, since they use separate normals and potentially have different texture coordinates). This gives us a total of $2{,}048 \times 4 \times 6$, or 49,152 vertices, just under the 65,536 vertex budget. Filling an entire chunk with blocks that have a more complex shape or with transparent blocks like glass will actually exceed the vertex budget (not a problem in the case of this project but for a game this is a case that will need to be handled, perhaps by spawning an additional mesh if 65,536 vertices is exceeded during mesh creation). The reason not to use smaller chunk sizes, such as 8^3 is that this would mean more meshes that need to be drawn (8 times as many in the case of going from 16^3 to 8^3), increasing the number of draw calls that need to be made, which significantly increases CPU overhead during rendering.

Therefore a chunk size of 16^3 blocks was chosen as the best middle ground between being too large (and potentially running into problems with overflowing the 16bit vertex buffer limit) and being too small (reducing rendering performance with too many draw calls). This matches the 16^3 render chunks employed by Minecraft and appears to be a common choice for other voxel world engines (for examples see the thread “After playing minecraft…” on the Unity3D forums at http://forum.unity3d.com/threads/after-playing-minecraft.63149/).

Some voxel world engines separate the block storage from the display chunks, for example choosing to store the blocks in a large flat ringbuffer, rather than as multidimensional arrays as part of the chunks. This has some performance benefits (cache coherence, less memory fragmentation) but at the cost of additional complexity. For the sake of simplicity the later approach was used, although block storage was changed from a multidimensional array to a flat array after I discovered that the C# runtime will perform three separate bounds checks when accessing a 3-dimensional array. Thus switching to a flat array and using custom logic to perform a 3d mapping turned out to be significantly faster.

One thing in common among all voxel world implementations is that the positions of voxels are always implicit, at least within their chunk. No positional data is ever stored with the individual voxels; rather the location in the storage array marks their position in the world (that is the location of a voxel is implicit). To access a voxel, the world manager first calculates the chunk position by dividing the position values by the `ChunkSize` constant (16 for this project). Rather than hardcode the value of `ChunkSize` a constant was used, allowing quick alteration if for some reason a different size would be needed. This is then used to look up the correct chunk. The chunks themselves are stored using the .NET Dictionary<T> class (which implements a hashmap), using their chunk-space position as a key. This allows fast retrieval of chunks as needed. The chunk-space position is the chunk number in each cardinal direction. So going right along the x-axis for example the first chunk is at (0,0,0), the second at (1,0,0), the third at (2,0,0) and so on. Once the correct chunk is found the remainder of the division is used to look up the specific block within the chunk block array. Storing the position data with every voxel would require an additional three `int32` values per voxel (12 bytes). A typical world generated by VoxBox for this project has about 5,000 chunks, or 20,480,000 voxels. Storing 12 extra bytes per voxel would mean 20,480,000×12 additional bytes or ~234Mbytes of memory without offering any extra utility. However, by explicitly storing the location of each chunk we never need to create or store chunks that are completely empty. This saves memory in a way similar to the sparse voxel octrees discussed in 2.3 Sparse Voxel Octrees (although the SVO does not use any explicit position referencing system).

4.1.3.1 Lighting

Lighting in VoxBox is a combination of dynamic and static lighting. Dynamic lighting is simply a low ambient light and a directional, shadow-casting light to emulate the sun. Because dynamic lighting is handled entirely by Unity3D, nothing further will be mentioned on that subject.

Static lighting is done with a crude global illumination approximation and baked into the chunk mesh vertices. The system used is similar to that employed in Minecraft.

First, an array is created for the light data. This is the same size as the block array for each chunk, so that one light entry is available for every voxel. Then, for the initial pass we trace sunlight from the top of the world down until a block with the `BlocksLight` property is set. This roughly simulates the direct illumination of the sun, assuming it is directly overhead (i.e. the 12o'clock position). Most importantly this is very fast because it is a simple linear march down the array storing the lighting values. A more sophisticated algorithm might utilize path tracing to simulate different sun positions but this would require additional calculation.

The next step is to spread the light out to simulate indirect illumination. Here a simple multi-pass algorithm adapted from Minecraft is used (Light - Minecraft Wiki). Lighting is restricted to 17 shades of illumination (from 0 for areas with no light to 16 for areas in full light). The function that calculates the light spread loops over the 3D array containing the light values 16 times, processing light values from 16 down to 1. During each pass the light entry is examined for every voxel cell. If the light value matches the current pass then we examine the six neighbour cells. For every neighbour that does not block light (that is the block property `BlocksLight` is not set) the light value of the neighbour is compared to that of the voxel being processed. If the neighbouring cell has a light value lower than that of the current voxel cell minus one, it has its value set to that of the current voxel cell minus one. For example say we are processing the pass for light values 8 and have found a voxel with a light value of 8. The neighbour to the left has a light value of 3 and contains a voxel that does not block light. Now this neighbour is set to have a light value of 7 (one less than the current pass being

processed). Now on the next pass processes cells with light values of 7, so this cell will be processed. It checks its neighbours and finds that the neighbour on the right has a value of 8 (this is the one that spread the light value 7 to this cell in the last pass) and therefore does not change that cell (of course the 5 other neighbours are examined as well). This way light spreads a maximum of 16 blocks from a primary source and diminishes along the way.

This is only a very crude global illumination approximation but it produces surprisingly good looking results while being very fast (11.79 seconds to calculate for 5214 chunks on an i7-4712HQ, or ~2.26ms per chunk). Moving the light spread function from C# to a C dynamically linked library showed a significant speed up even over that (2-3x speed increase) but introduced errors into the lighting that were not resolved. These errors may be due to a bug in the C code that I wrote, or due to some oddity in the way that multidimensional arrays are marshalled to and from native code by the C# runtime. Either way this should be resolvable and would offer a good increase in speed for this function. Due to its non-critical nature, it was left as is and time allocated to more important aspects of the project. However, the native C version of the function was used during most of the testing because it is so much faster and the more correct looking C# implementation was later used to produce cleaner screenshots. Figure 24 below highlights an example of where lighting problems occurred with the native implementation of the lighting algorithm. The shadow on the castle wall clearly looks incorrect in several areas (a particularly bad area is highlighted with a circle and arrow).

Figure 24 – Lighting Anomaly

The final part of the lighting calculation is to apply the lighting values to the vertices of the polygon mesh that is generated from the voxel data. We cannot apply the light value calculated for a voxel cell to the vertices generated for that voxel, since most voxels block light and so would have a value of zero. Instead what we want to know is the amount of light hitting a voxel face from the surrounding voxel cells. To get smooth transitions from light to dark we take an average of the four light values in the voxel cells surrounding the vertex in the direction of the voxel face. Figure 25 illustrates this process. In the image the solid cube is the voxel we are currently calculating lighting for. The red lines show the current face. The red sphere is the current vertex and the red arrow shows the vertex normal direction. The four translucent cubes show the four voxel cells that are sampled for lighting information for the current vertex.

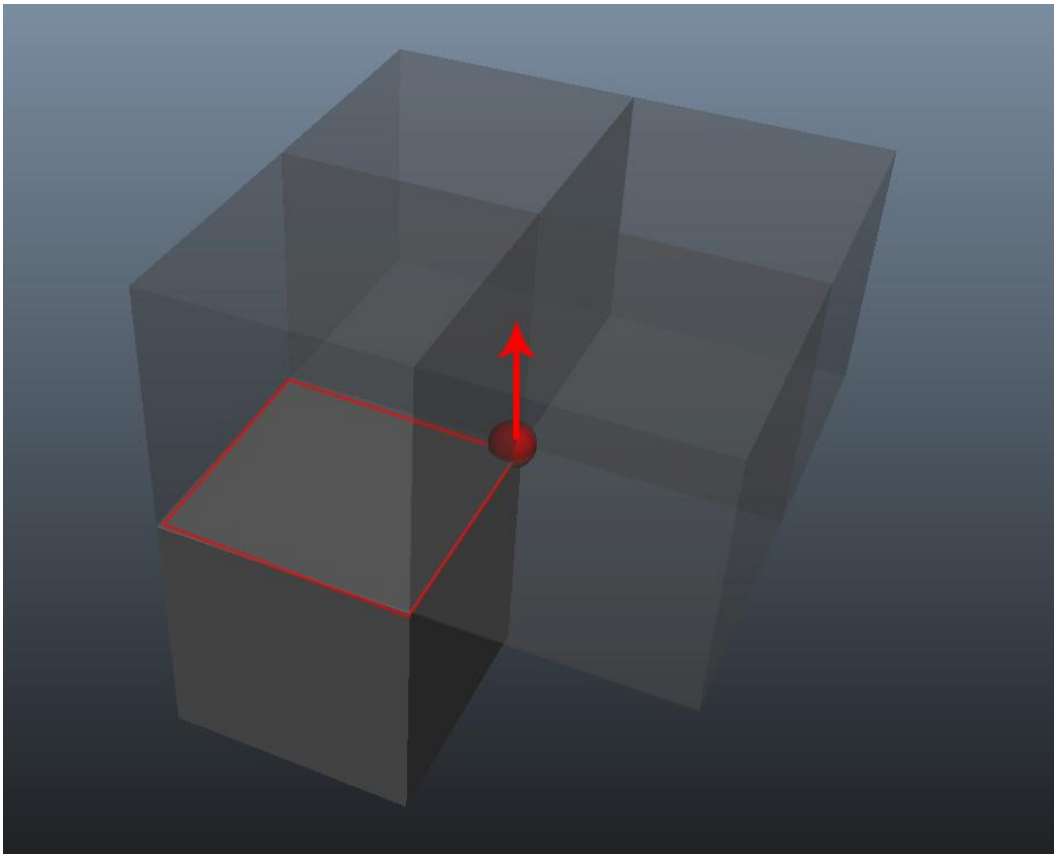


Figure 25 – Smooth Lighting Calculation Illustration

A side effect of calculating the lighting this way is that vertices in concave areas are darker, adding an ambient occlusion[6] like effect. This can be seen in Figure 26, where the surface surrounded by blocks is clearly darker than the open surfaces. This is a subtle effect that makes the voxel world look more aesthetically pleasing, while being cheap to calculate (it only needs to be computed when the mesh is created).

[6] Ambient occlusion simulates self-shadowing by casting rays from a surface and darkening the area for any rays that hit a surface within a certain distance.

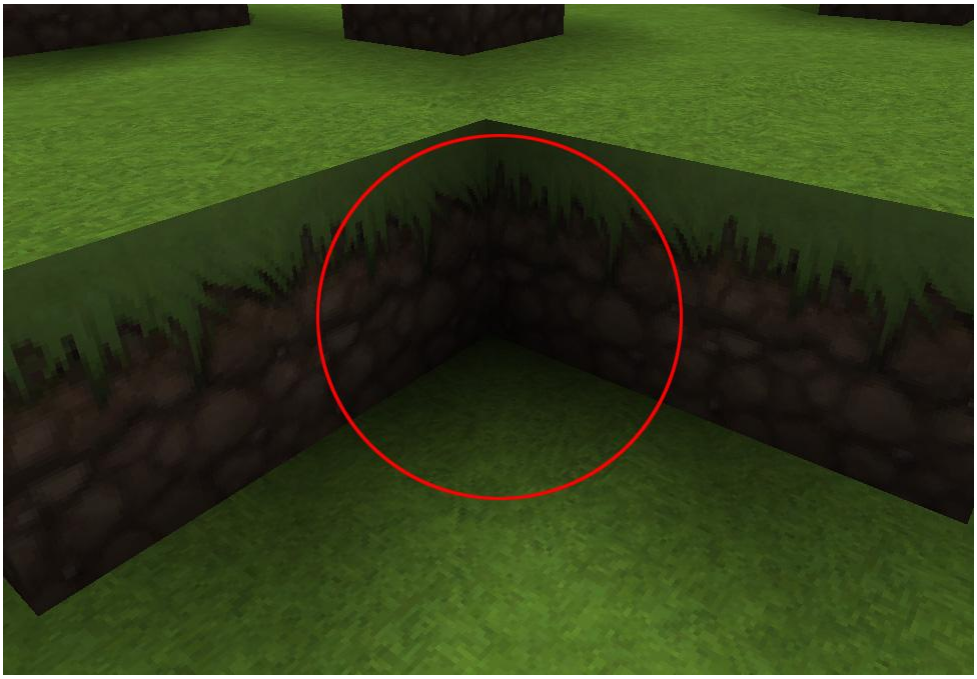


Figure 26 – Fake Ambient Occlusion

4.1.3.2 Conversion to Polygons

In order to display our voxel world using a traditional GPU we must convert the voxel data to a polygon mesh. There are other possibilities, such as raycasting directly into a voxel data structure as described by Crassin et al in their paper on the GigaVoxels rendering system (Crassin, Neyret, Lefebvre, & Eisemann, 2009). However, that approach is more appropriate for much more dense voxel data, such as that from Lidar scanning of real world objects or environments.

A naïve approach would generate polygon mesh data for every non-empty voxel cell. This would quickly create a more polygons than can be reasonably displayed even by a powerful GPU and would be a massive waste. Instead we only create mesh data for exterior (exposed) faces. Figure 27 shows two chunks moved slightly apart, showing how the interior faces are not generated (red highlighted area).

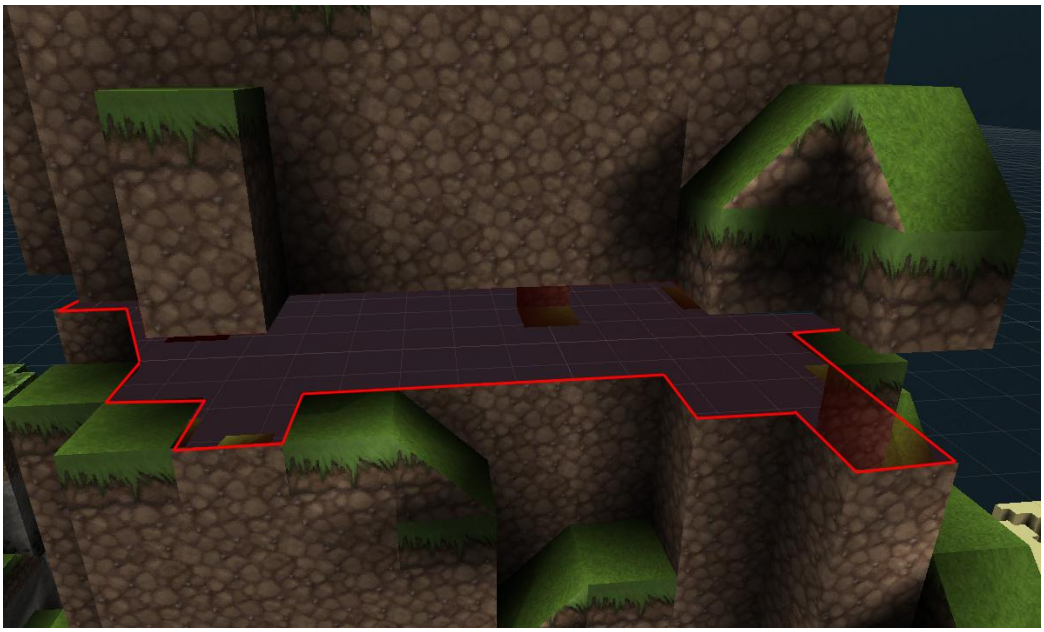


Figure 27 - Two Chunks Apart Showing Hidden Faces

Neighbour voxels are checked during chunk mesh generation and for any that `IsTransparent` is set to false no face is created.

We want to support more complex shapes at this point than just cubes, in the same way that Minecraft does (this was discussed in the literature review in section 2.5 Case Study: Minecraft). In order to support many different types of geometry for voxels, without a large switch statement in the mesh generation code, the concept of a geometry builder was introduced. This is discussed in the next section: 4.1.3.3 Geometry Builders.

4.1.3.3 Geometry Builders

Creating the `IGeometryBuilder` interface made it simple to control the type of geometry created for each non-empty voxel. The interface has only a single function definition – `BuildGeometry()` – that is called once for each non-empty block during the mesh creation process. The function is passed a reference to the chunk object that is calling it and uses this reference to add vertices and indices to the chunk. The vertices and indices are stored in a linked list within the chunk until `BuildGeometry()` has been called on every block for that chunk. Once complete, the vertex and index data is placed into a Unity3D `Mesh` object on the main thread. Calculating all of the vertex and index data for the terrain mesh is done on the background thread, but because all Unity3D functions must be called on the main thread the final step of creating the actual Mesh object must

be done on the main thread (for more on the threading system used refer back to section 4.1.1 Introducing VoxBox).

The geometry builders implemented for this project are as follows:

- `BoxBuilder` – Standard builder used for all regular cube-shaped blocks.
- `MicroBlockBuilder` – Described in more detail in section 4.1.3.4 MicroBlocks below.
- `WaterBuilder` – Used for water blocks. Only the top faces are created for these and the indices are stored in a separate buffer, making use of the submesh functionality in the Unity3D `Mesh` object. The submesh is rendered in a separate draw call, allowing it to use a different shader from the rest of the terrain and be translucent.
- `WedgeBuilder` – Creates a ramp shape used to smooth out the grassland terrain (example output is shown in Figure 28). These are used to smooth out the terrain a bit (terrain smoothing is discussed in section 4.2.10 Smoothing the Terrain).
- `WedgeFillBuilder` – Creates a single triangle that can be used to fill the sides of the geometry generated by the `WedgeBuilder`. Also used as part of the terrain smoothing algorithm (for an example see Figure 29).
- `XBuilder` – Creates an X shape on the block. This is used for grass.

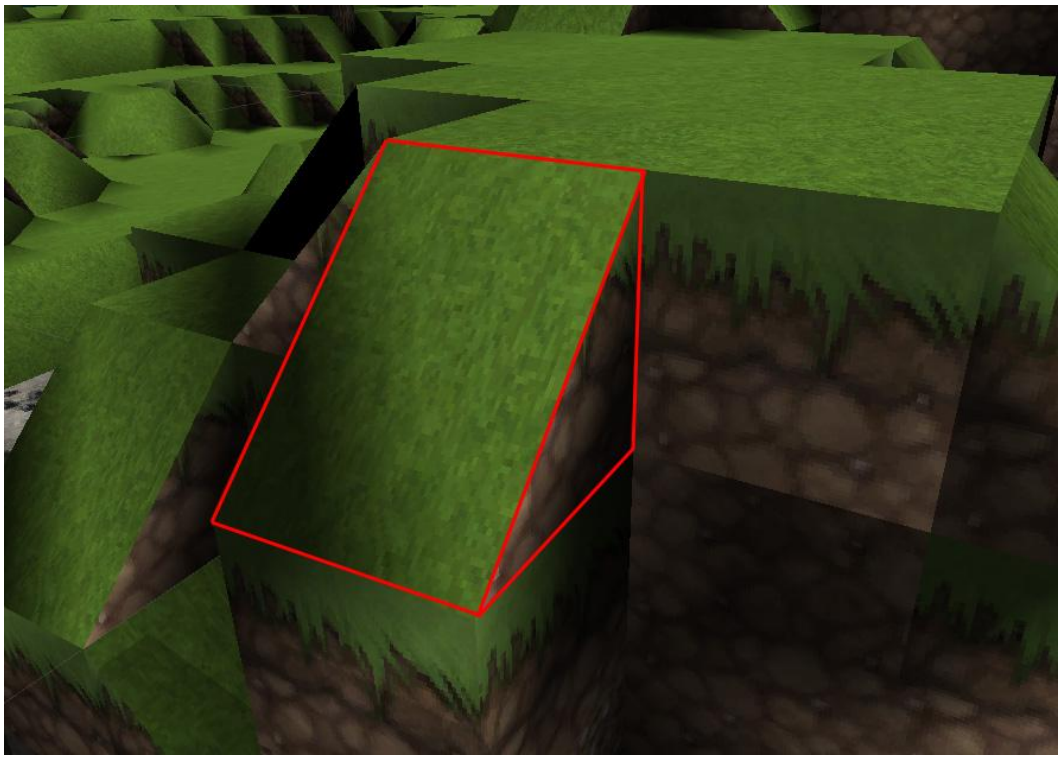


Figure 28 – WedgeBuilder Output

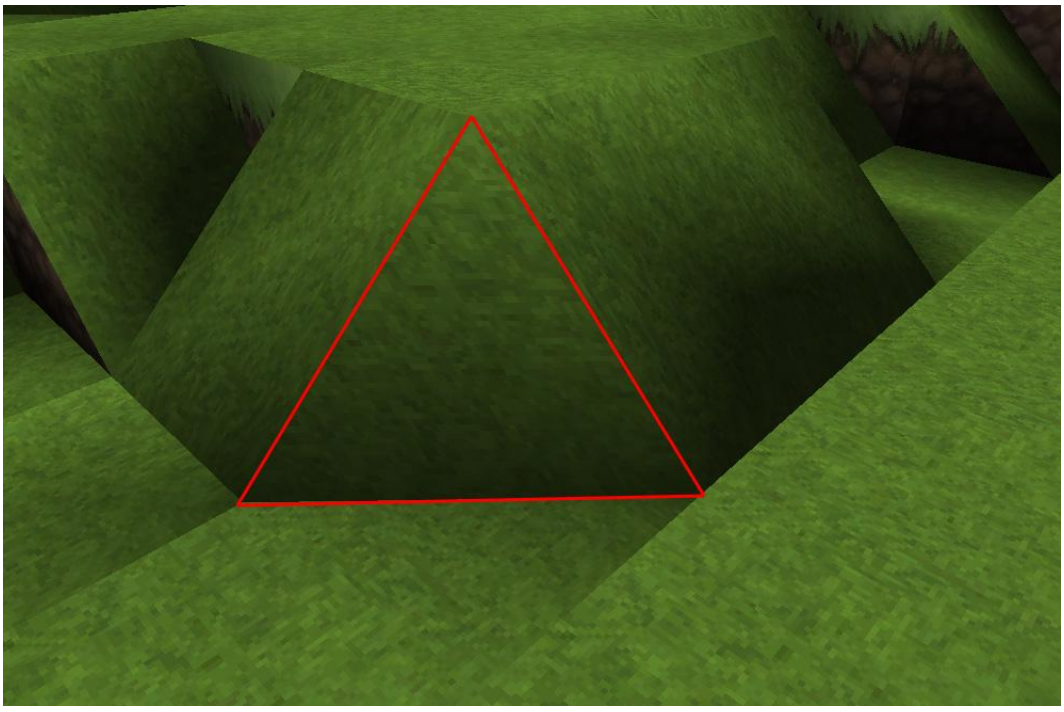


Figure 29 – WedgeFillBuilder Output

4.1.3.4 MicroBlocks

As shown in the literature review (in section 2.5 Case Study: Minecraft) it is possible to create a more detailed look to the world with special blocks like stairs or fences. In order to be able to add more complex shapes to the voxel world

easily, a special geometry builder was added to VoxBox: the `MicroBlockBuilder`. This builder takes a 3D array of Boolean values. This can be any size as long as all 3 dimensions are the same size. It essentially subdivides the voxel into smaller cubes $(\frac{1}{n})^3$ in size (where n is size of each of the dimensions in the array). These smaller cubes all share a single texture set (one texture per face of the voxel cube) for the sake of simplicity. Texture coordinates and lighting information is simply interpolated across the surface of the voxel for each micro block.

Using micro blocks like this allows easily adding new types of geometry like stairs, slabs, slits, or poles, all without having to write a new custom geometry builder for every new kind of geometry. As long as it can be constructed from smaller cube shapes it can be added simply with a `MicroBlockBuilder` and an array of Boolean values indicating which sub-blocks should be filled and which should be empty.

For this project micro blocks were used to add the crenellated battlements to the tops of the walls, and to create the stairs for the walkways on the walls.

4.1.3.5 Layers

Part way through this project it became apparent that it would be useful to be able to make non-destructive changes to the voxel world (for example, the ability to toggle the visibility of the castle to be able to check it against the underlying terrain). Inspired by the layer system in 2D drawing programs like Adobe Photoshop™ I decided to implement something similar in VoxBox. This required some changes to how voxels are stored. Instead of being stored in an array as part of the chunk they belong to, they were moved into a separate data structure called a VoxelLayer. Each chunk now contains two layers: a “background” layer where the terrain is drawn and a top layer where the castle is drawn (adding more layers would be trivial at this point but two were sufficient for this project). By toggling visibility on the top layer we can quickly see the terrain with or without the generated castle (for example the screenshots in Chapter 5: Demonstrations were created this way, i.e. one with just the terrain and one with the castle in the terrain).

In addition, a new voxel type was added: the “Empty” voxel. This like the “Air” voxel type also marks a voxel cell as empty. On the background layer empty voxel cells always use the “Air” type. On the other layers the two types are distinguished thusly: “Air” voxels force the voxel cell to be empty, while “Empty” voxels are treated the way that transparent pixels would be in a 2D drawing program, i.e. they are ignored. When a voxel is accessed the top layer is queried first. If the position contains any type other than the “Empty” type then that voxel type is returned. If the voxel cell contains an “Empty” voxel then the next layer down is queried with the same process until a non-“Empty” voxel type is encountered. Since the background layer cannot contain voxels of the “Empty” type a valid voxel is guaranteed for each lookup. This is easiest understood by comparing it to layers in a 2D drawing package. Typically you start with a white page (this would be a background layer with “Air” voxels). When you add an empty layer the picture stays the same (like adding a layer with “Empty” voxels). Now imagine you draw a black square on the background (like adding “Dirt” voxels to the background layer), then you cover this square with a white square on the top layer. This way you end up with what looks like a blank page once more (this would be like filling the volume of “Dirt” blocks with “Air” blocks on the top layer, giving you an empty voxel world). Finally, if we toggle the visibility of the top layer we can reveal or hide the black square (the same is true for our voxel layer).

Layers that are entirely empty (either background layers where all blocks are set to “Air” blocks, or regular layers with all blocks set to “Empty”) have their voxel arrays set to `null`. This means that empty layers, and indeed empty chunks, use very little memory. When queried the empty layers simply return “Empty” or for background layers “Air”. As soon as a voxel is set to a non-empty value the voxel array is created and the voxel is set in the array.

To recap: layers function in a way similar to layers found in many 2D drawing packages. By drawing the terrain on the background layer and the castle on a higher layer we can easily toggle a scene between displaying just the terrain, and showing the castle in the terrain.

4.2 Terrain Generation

The common approach to randomly generating any sort of terrain is to make use of one or more coherent noise functions (for more information on coherent noise see section 4.2.1). This is also the approach that VoxBox uses to create its terrain. It uses several different noise functions chained together to generate a small variety of terrain. Each of the noise functions used is described in this section, followed by a description of how they are combined to reach the final result.

The terrain generated needs to be complex enough to present an interesting challenge for castle placement. It also must be capable of a good amount of variation. As discussed in section 4.5 Castles, the castle generation algorithm is deterministic, so any variation in the castles will come from the underlying terrain. With this in mind the VoxBox terrain generator was built to create three broad types of terrain: mountains, rolling hills, and flat grassland. This was enough to provide sufficient variation in the terrain and also provide a challenge for the castle placement and construction.

Terrain generation consists of three major steps. First, a heightmap is created for the terrain by combining several noise functions. Secondly, a combination of the heightmap and several more noise functions is used to calculate a block type for every block in the world. Lastly, the terrain is processed to add some extra details like trees, water and terrain smoothing.

This project makes use of Libnoise, a library that provides a number of different coherent noise functions. The first part of this section covers Libnoise and what coherent noise is. It then follows with descriptions of each of the noise functions used and example output from each. How these noise functions are combined to create the heightmap used in the terrain generator is covered next in section 4.2.6 Creating the Heightmap. The terrain generation section concludes with a description of how blocks are calculated and descriptions of the final steps that cap off terrain generation (adding in trees, water and some terrain smoothing).

4.2.1 Libnoise

Rather than implement each of the needed noise functions and a system for combining them myself I made use of Libnoise (http://libnoise.sourceforge.net), a library designed specifically for this task. Libnoise comes with a variety of noise

functions and modules for modifying and combining the outputs of those functions. This is perfect for creating the sort of noise generation network needed to create interesting terrain. Libnoise contains more noise functions than needed for this project; however those that were used are described in more detail below.

Libnoise generates “coherent” noise, that is it creates a type of pseudorandom noise where neighbouring values are related in some way, as opposed to non-coherent noise where all values are independent of each other. Coherent noise when given the same input (i.e. when a location is sampled) will generate the same output value, a small change in the input will result in a small change to the output, and a large change in input will lead to a random output (Bevins, 2005c).

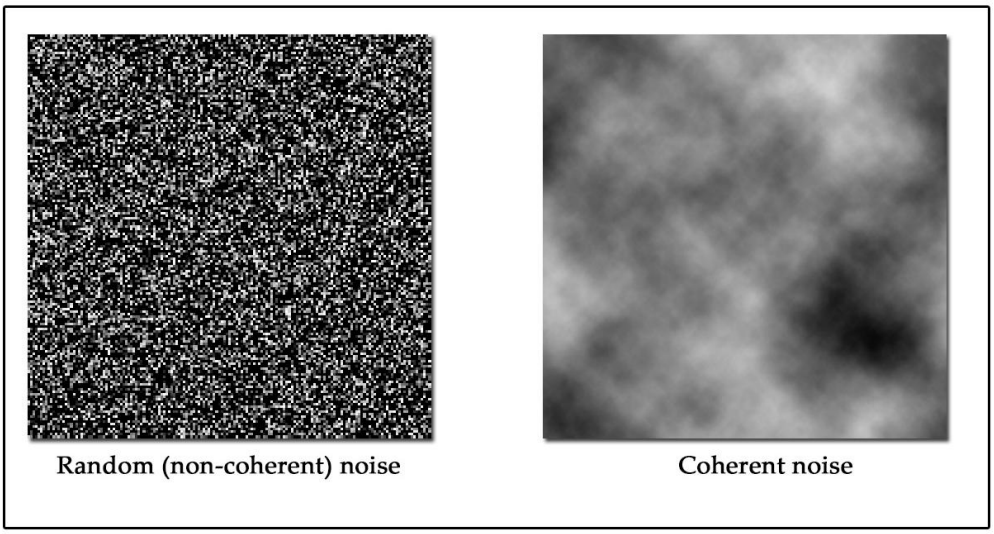


Figure 30 - Random Noise vs Coherent Noise

Figure 30 above shows the difference between completely random and coherent noise. The smooth gradients that coherent noise creates are well suited to creating heightmaps for terrain generation.

Note that all Libnoise generator modules output values in the range -1 to 1 by default, although this can be manipulated with additional modules (covered after the noise modules in this section).

4.2.2 Perlin noise

One of the most well known noise functions, Perlin noise is a gradient noise generator capable of generating a variety of looks such as clouds, glass, water and more (Perlin, 1984) (Perlin, 1985) (Perlin & Hoffert, 1989). At its core it

functions by creating random values and then interpolating between them to create gradients (hence it is called gradient noise). An example of Perlin noise is shown in Figure 31 below.

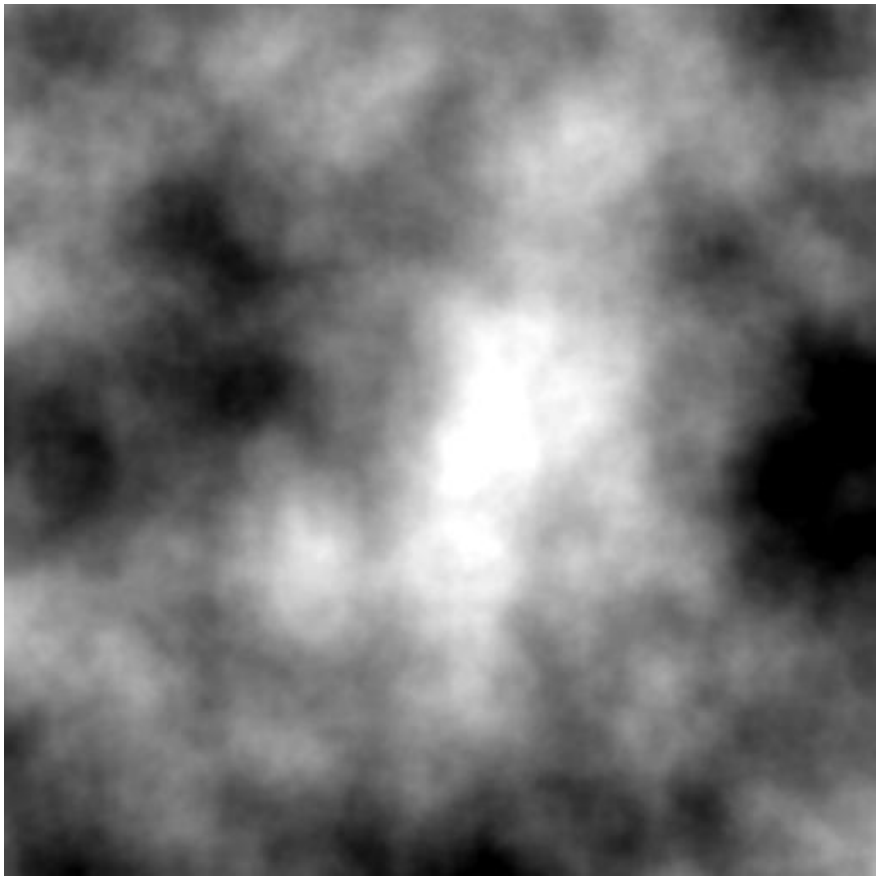


Figure 31 – Perlin Noise

Perlin noise can be controlled with two key variables: frequency, and number of octaves. Frequency affects the spatial frequency of the noise function. This alters how rapidly changes occur over any given portion of the function space. This effect can be seen in Figure 32 below, showing the same Perlin noise function with a frequency of 0.005, 0.05, and 0.5.

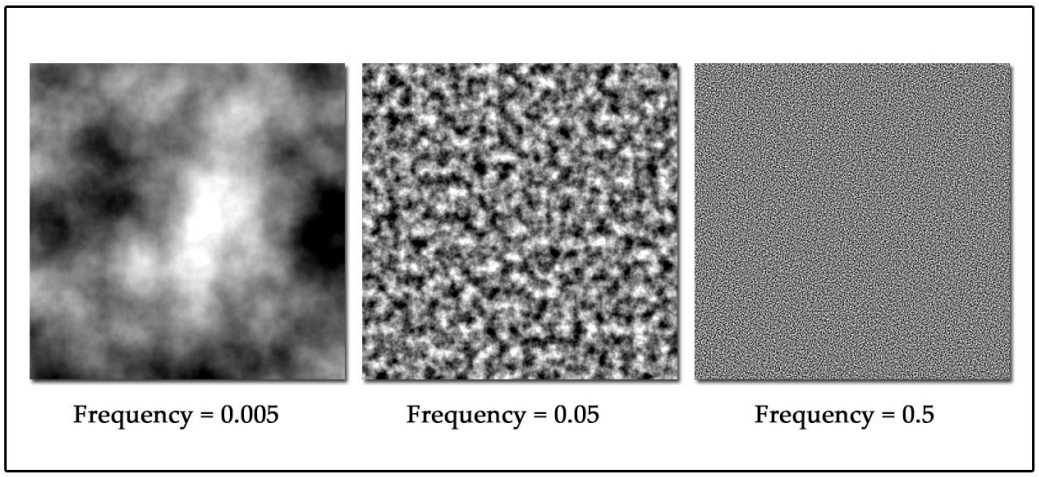


Figure 32 – Effect of Changing Frequency on Perlin Noise

The number of octaves changes the amount of detail generated by the Perlin noise function. Perlin noise works by adding together multiple passes over the noise function at successively higher frequencies to generate additional detail. The number of passes done is controlled by the octave number of the function. It is important to note that increasing the number of octaves increases the computational complexity of the function (i.e. it will take longer to compute the noise values). Figure 33 below shows the result of running Perlin noise with octave values of 2, 4, and 8. Note the increased complexity in the image as the number of octaves is increased.

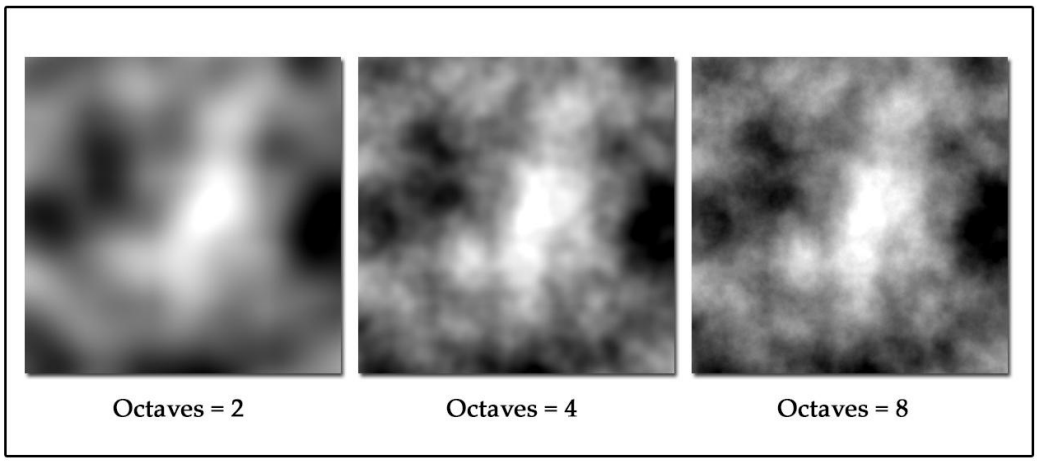


Figure 33 – Effect of Changing Octaves on Perlin Noise

Perlin noise is used for most of the terrain generation process, including selecting which type of terrain should be generated at any given point. The hilly terrain uses a slightly modified version of Perlin noise that Libnoise calls billow noise. According to the Libnoise documentation this is identical to Perlin noise except

that every octave is modified by an absolute-value function (Bevins, 2005a). This produces a billowy look suitable for clouds or in the case of this project the heightmap for rolling hills. An example of what this looks like is shown in Figure 34.

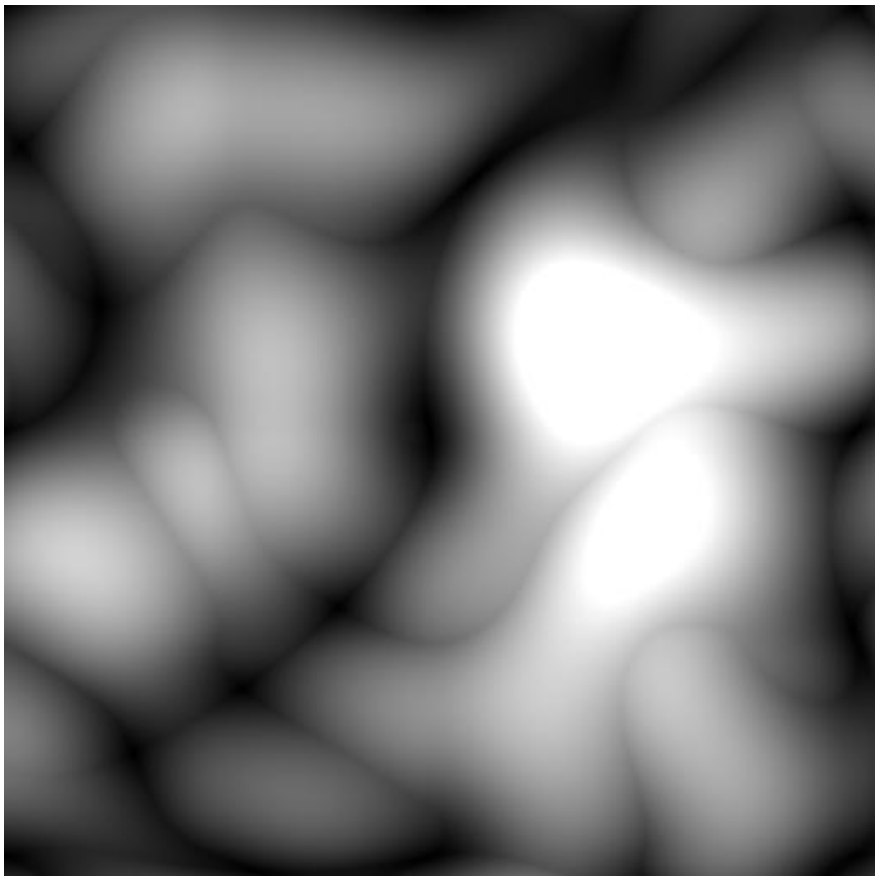


Figure 34 – Billow Noise

4.2.3 Ridged Multifractal Noise

Ridged multifractal noise in Libnoise is generated by a similar process to Perlin noise but octaves use persistence values derived from previous octaves. This creates a feedback loop that creates ridge-like formations in the output (Bevins, 2005b). This can be seen in Figure 35 below. Note how the white areas in the image form long lines with very bright areas along their centre. These are the ridges that give ridged multifractal noise their name.

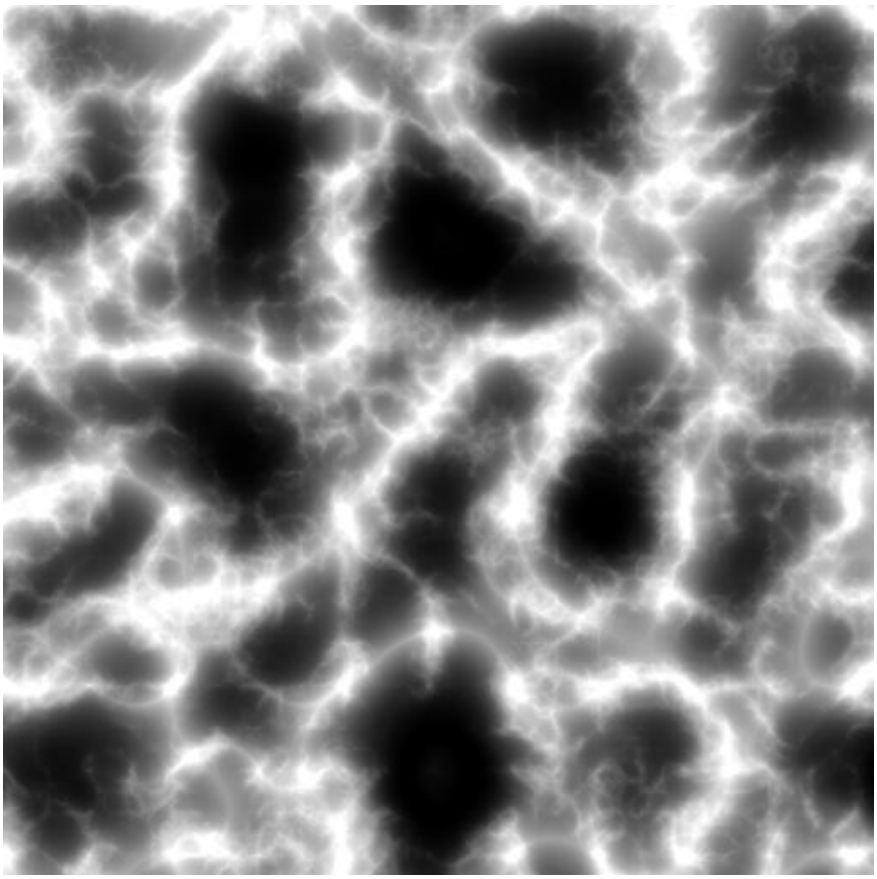


Figure 35 – Multifractal Noise

The terrain generator uses ridged multifractal noise to generate mountain areas and to carve out cave systems.

4.2.4 Vornoi Noise

Vornoi noise (otherwise known as Worley or cellular noise) is a point-based noise generation algorithm. The output is generated by scattering random points across a plane and then computing where neighbouring points are equidistant. The plane is then divided up by lines along these equidistant regions, creating a Vornoi diagram; partitioning the space into cellular regions (Worley, 2002). This kind of cellular noise is useful for generating a variety of effects such as the look of lizard scales or flagstones. For this project the Vornoi noise was used to generate the field areas, however the effect of this in the final computed heightmap is very subtle. It is simply used to break up the field areas slightly so that they would not be completely flat. An example of Vornoi noise is shown in Figure 36 below.

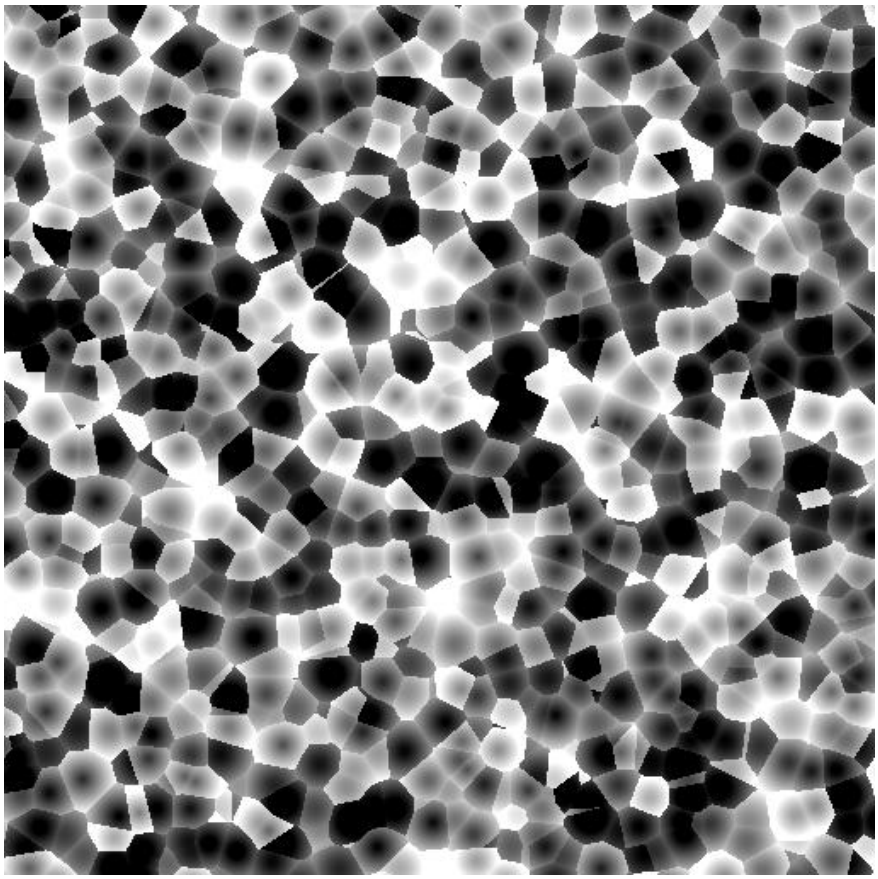


Figure 36 – Vornoi Noise

4.2.5 Additional Libnoise Modules

In addition to the noise generator modules a number of other utility Libnoise modules were used to connect everything together. These are mostly very simple and therefore will only be given a brief explanation here.

- Scale Bias module – multiplies the output of a module by a scaling value and adds the bias value to it. Used to control the height of mountains for example.
- Select module – takes two input modules and a control module as input. The output becomes either the value of the first or second module, depending on if the value returned by the control module falls within a certain selection range that is set when the module is created.
- Turbulence module – A pseudorandom displacement of an input value. Uses Perlin noise modules to displace the x, y, and z values passed into the module by some amount.

4.2.6 Creating the Heightmap

The first step is to combine multiple noise modules to generate a heightmap that can be used by the terrain generator and that creates areas of plains, hills and mountains. This process of adding and rescaling noise functions is also known as multifractal construction (Musgrave, 2002). A powerful feature of this approach is that individual points can be evaluated without reference to each other. The context-free nature of this method is what makes it possible to generate new chunks of the world on the fly and enables a voxel world to grow almost limitlessly as a player explores. For this project only a small section of the world is created but this approach demonstrates how a real voxel world game would generate its terrain.

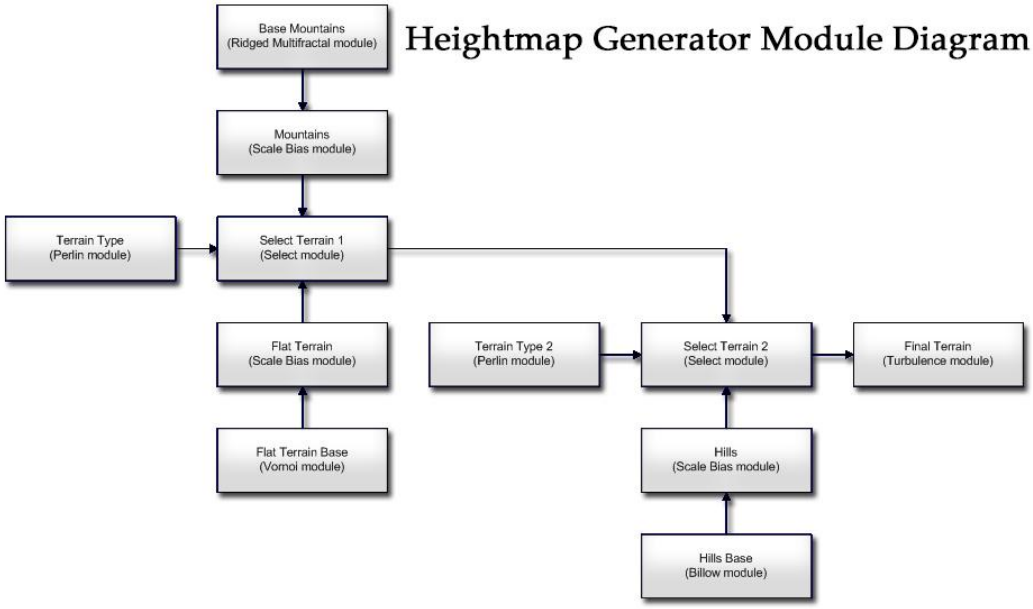


Figure 37 – Heightmap Generator Diagram

The diagram above (Figure 37) shows how the various Libnoise modules are combined to output the final terrain heightmap. For an example heightmap generated by this process see Figure 38 below. Note that due to the mapping from height values to greyscale; some of the values were clamped resulting in a loss of detail in the extremely high (white) parts of the image.

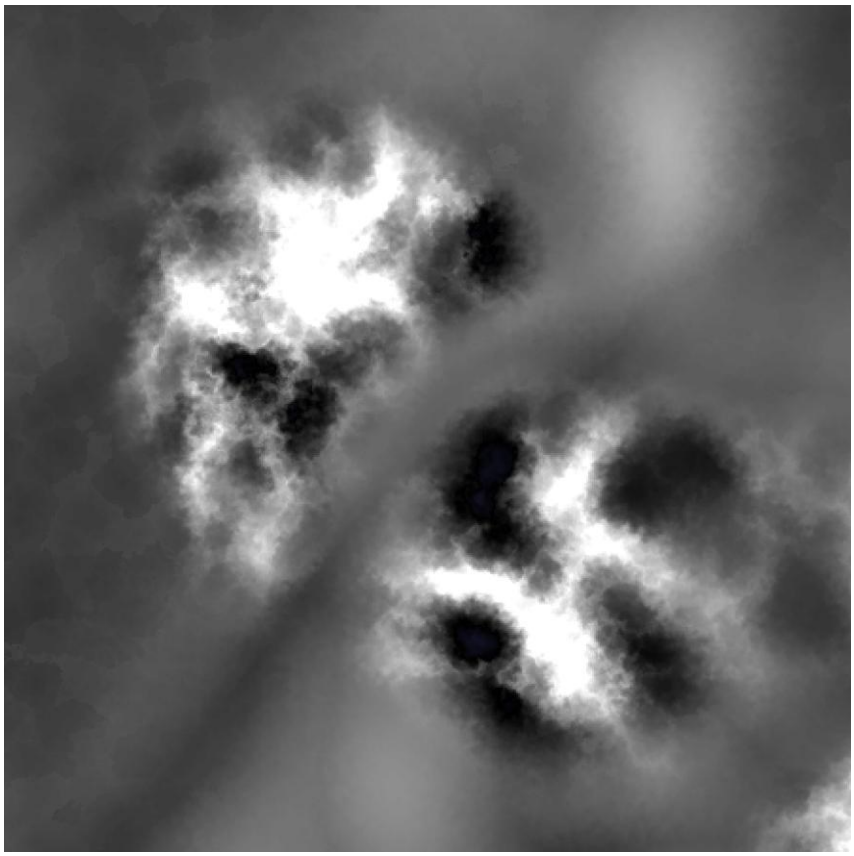


Figure 38 – Example Heightmap

During the development of the castle generation algorithm (described in detail in section 4.5 Castles) it was found that mountain areas were far too low (essentially just craggy hills when compared in scale to the castles) (see Figure 39). The castle being almost as tall as the mountains made it hard to properly evaluate placement and did not provide the correct sense of scale. Because the chunk system (described in section 4.1.3 Chunks) was designed to allow for flexible world height this was as simple as adjusting the heightmap generator to increase the height of the mountain areas by 4 times (see Figure 40). This created a much more realistic backdrop for the castles, and provided better terrain to test the castle placement against. Because the variation in the castles is driven entirely by the terrain (explained in section 4.5 Castles) this was an important change. Stronger variation in the terrain also meant stronger variation in the castles produced.

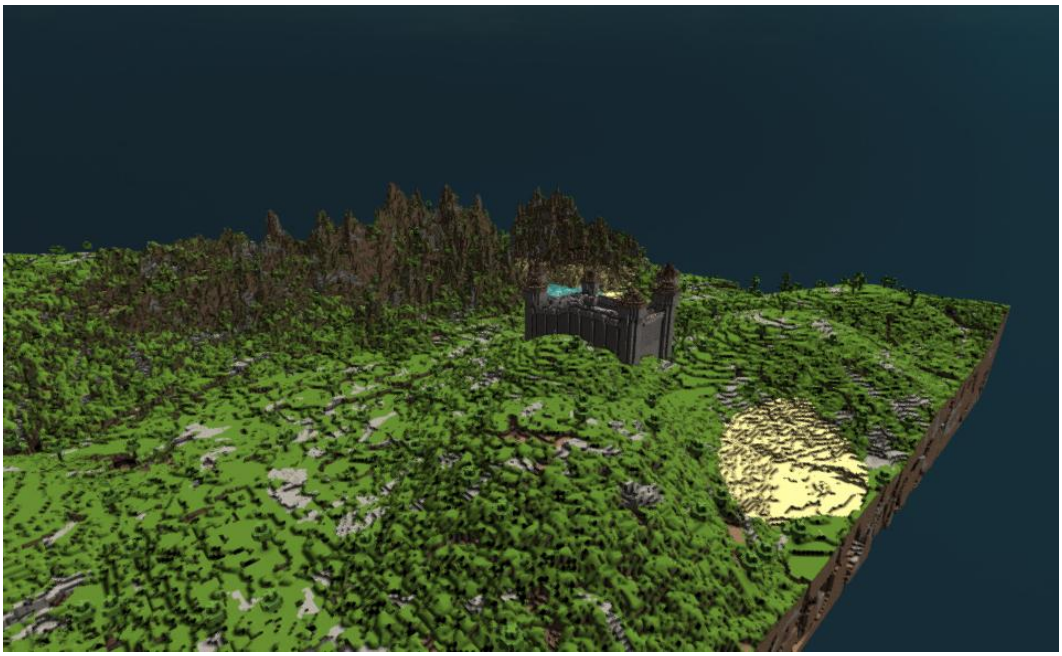


Figure 39 - Terrain and Castle Before Mountain Height Adjustment

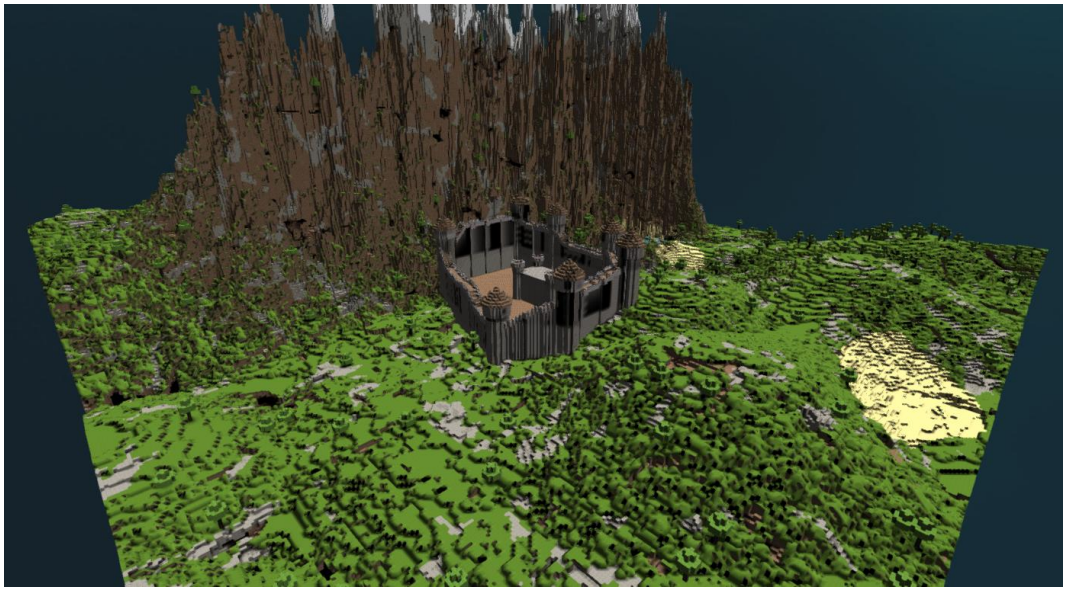


Figure 40 - Terrain and Castle After Mountain Height Adjustment

4.2.7 Calculating the Blocks

With the heightmap created the first major step in creating terrain is complete. The next step is to compute the terrain blocks using a combination of the heightmap and several additional noise modules.

In addition to the heightmap, three more Perlin noise modules are employed in the terrain generation process, one each with a frequency of 0.005, 0.05 and 0.5. This gives us a low frequency noise function for large-scale features, a medium frequency function for mid-scale features and a high frequency noise module to

use for small scale details. They will be referred to as low, medium, or high frequency noise modules for the remainder of the section. Furthermore, a multifractal noise module is used in the terrain generation process (separate from the one used to generate the heightmap).

Output from each of the noise modules is cached ahead of time so that accessing them multiple times does not incur a recalculation. This is done because calculating a noise function is expensive and some values will be used multiple times. This was not needed during heightmap creation because in that case every value is queried just once.

All the actual terrain calculation is done in a function in the terrain generator called `CalculateBlock()`. This function simply takes a position of a voxel cell as an x, y and z value, and returns a block type. Several numbers are used in this process, usually as threshold values against a noise function. These were mostly arrived at by a process of trial and error to achieve the desired[7] look of the terrain. Where applicable any reasoning behind the values is mentioned. The important thing to remember is that values from noise functions are generated in a range of -1 to 1, so for example, if a threshold is set to greater than 0, that means it will be true about half of the time. If set to greater than 0.5, it would be true about a quarter of the time, and so on. Keep in mind that when a block is calculated the function immediately returns with that value. That means that block calculations that happen further on in the process become increasingly less likely to be reached in the first place (for example if the first calculation had a 50% probability of generating a block A and the next calculation has a 50% probability of generating a block B, then 50% of blocks would be of type A, but only 25% of blocks would be of type B, i.e. half of the remaining 50%).

The operation of this function is as follows (in order of evaluation):

[7] The desired look for this project was one similar to the terrain in Minecraft, which can perhaps be best described as heightened or compressed reality. That is that features should be recognizable as terrain that exists in the real world (mountains, valleys, forests, deserts, and so on) but compressed into a smaller world size so to provide plenty of variation to a player within an accessible radius.

1. If the y-value of the block is less than -50 plus the value from the high frequency noise module return a “Stone” block. This represents the lowest part of the world we care about. Anything lower simply becomes stone.
2. If the y-value is above the heightmap and above sea level (see 4.2.7 – Water) return a block of type “Air”. If it is above the heightmap value but below sea level return a “Water” block instead.
3. Create caves. This is computed by getting a value from the multifractal noise generator and returning “Air” if the value is above a certain threshold. This threshold is set to 0.8 if the y-value for the current block is above 0, otherwise it set to 0.4. Recall that values are generated in a range of -1 to 1. This makes caves quite rare above 0 and more numerous below 0 in the world.
4. Calculate a depth value. This is simply the value from the heightmap minus the y-value of the block being calculated.
5. Next deserts are added to the mix. If the depth value calculated in the last step is less than 20, and the heightmap value for this block is lower than 15, and the value from the low frequency Perlin noise module is greater than 0.42 the block qualifies as a desert block. What this means is that desert areas are only generated up to a depth of 20 and only in low-lying areas (no mountains made out of sand). If the depth is less than 6 a “Sand” block is returned, otherwise a “SandStone” block is returned. So all desert areas are simply a layer of sand, followed by a layer of sandstone.
6. Now stone blocks are calculated. This uses the ridged multifractal noise module. If it returns a value above 0.66 a “Stone” block is returned. This creates streaks of stone throughout the world to break up the dirt.
7. Next are the mountaintops. These consist of stone blocks, covered with snow blocks. It was important not to create a simple cut-off height for snow, otherwise an unnatural straight line could be seen on mountains where the snow begins. For this reason a SnowMin and SnowMax value were used and compared against the medium frequency Perlin noise module. SnowMin was set to 150 and SnowMax to 170. So no snow is ever generated below 150 and it is always created above 170. Between those values it is sometimes created, depending on the value of the medium frequency Perlin noise module. In all cases the “Snow” block is

only returned if this block is the top block (i.e. the y-value of this block is equal to the value of the heightmap for this location), otherwise “Stone” is returned, so that snow only covers the mountaintops in a single layer.

8. Next, the top layer of what remains is calculated. This is either grass, or tree stumps used to generate the trees later on (see section 4.2.8 Trees for more detail). First we check if this is the top block (if the y-value is equal to the heightmap value for this position). If it is, then there is a chance that a tree stump will be created. Whether a tree stump is generated or not is determined by a random number generator combined with a low frequency Perlin noise generator. This creates forests that thin out to the edges. The exact formula for this can be shown as a table:

Noise value threshold	Tree stump chance
>0.7	1 in 32
0.5-0.7	1 in 128
0.3-0.5	1 in 256
<0.3	1 in 1024

Table 1- Tree Generation Chance Values

9. The penultimate step is to calculate gravel blocks. This uses the ridged multifractal noise module and returns a “Gravel” block if the value is above 0.77. This might seem like a high value (almost 1 in 8) but remember this is only reached if none of the other conditions thus far have been met.
10. Finally, if none of the other conditions were met a “Dirt” block is returned.

For an example of what this all looks like when combined into a final terrain see Figure 41 below:

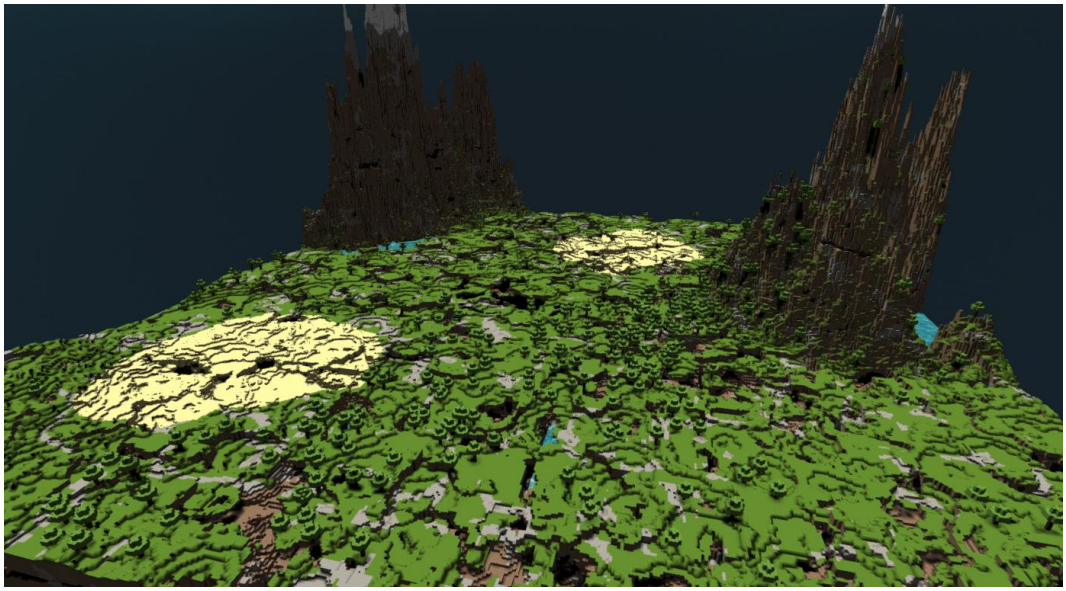


Figure 41 – Example Terrain

4.2.8 Trees

Trees are generated by placing seed tree stump blocks randomly instead of "Grass" voxels as explained in section 4.2.7 Calculating the Blocks. The actual trees themselves are generated by passing over all chunks after terrain generation has completed and finding all the tree stumps ("OakLog" blocks) placed in the world, then generating a tree of random height (5-10 metres) on top of it. This process simply involves placing more "OakLog" blocks on top of the seed block up to the desired height and then laying out leaf blocks around the top of the tree in an approximately hemispherical shape. An example of what these trees look like can be seen in Figure 42.

Figure 42 – A Tree Created by the Terrain Generator

For this step care needs to be taken to process the chunks from top to bottom. Initially this step was performed by processing chunks from bottom to top as with most other operations. This resulted in an interesting bug when trees crossed chunk boundaries and their tops were then processed as if they were seed tree stumps. The result of this bug can be seen in the figure below (Figure 43).

Figure 43 – A Tree Affected by the Tree Processing Bug

This was not a major issue, but it did cause some trees to grow far larger than desired. It does demonstrate how small changes to the tree generation algorithm could be used to generate a variety of different trees.

4.2.9 Water

For water creation there is a sea level constant specified in the world generator. For the purpose of this project it was initially set to 0 but moved to -5 to prevent some low lying areas from being flooded. Water is added to the world in two steps. First, during world creation water is created in voxel cells that are below sea level but above the terrain heightmap. This happens before the step that adds caves to the world so that caves below sea level are not automatically flooded.

However, this means that it is possible for there to exist areas where a cave system intersects the ground level in a place with water. Once the initial world terrain generation is complete all chunks are scanned for water blocks and a recursive algorithm is employed to fill the water to empty blocks around and below the water blocks. This ensures that any areas where water blocks and cave systems intersect are flooded correctly.

4.2.10 Smoothing the Terrain

In order to offset the blockier look of the castle from the terrain a bit better some simple smoothing is applied to the terrain. All this does is examine the neighbours of grass blocks and select to place ramp-shaped blocks for certain configurations in a way inspired by the marching cubes algorithm, only simpler (marching cubes was discussed as part of the literature review in section 2.4 Voxel Smoothing Techniques). The effect of the terrain smoothing step can be seen in Figure 44.

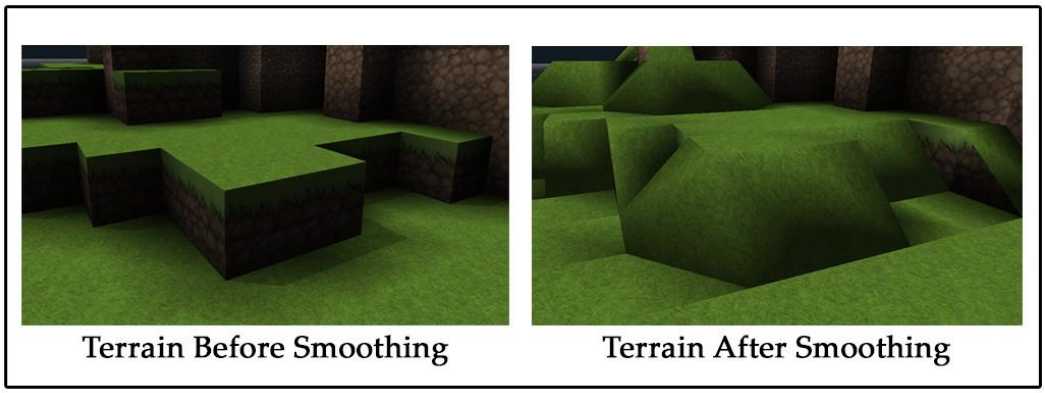


Figure 44 – Effect of the Terrain Smoothing[8]

[8] Note that the terrain after smoothing exhibits some shading artefacts. This is due to the additional geometry (the ram-shaped blocks) being placed on top of the existing terrain geometry. Some extra work could be done to detect these areas during mesh creation to avoid these errors.

4.3 Drawing with Voxels

With terrain generation complete we take a look at how we can add other things to the voxel world (such as the castles described in the upcoming section 4.5 Castles). Constructing buildings in a voxel world is remarkably similar to raster drawing on a 2D surface, only in 3D instead. In fact most of the time drawing in just two dimensions with the third fixed is all that we need. A number of 2D drawing algorithms were adapted and implemented for this purpose. These make use of a brush system, much like you would find in a 2D drawing system such as the one described in (Foley, van Dam, Feiner, & Hughes, 2001). This turned out to be very useful for a number of tasks outside simple drawing as well.

All of these drawing functions operate along the x and z axis of the world (i.e. horizontally). A y-coordinate is also supplied and this is kept fixed throughout the drawing process.

To use the drawing functions first a brush is set in the drawing object and then one or more of the drawing functions are called. Where a voxel is to be output by the drawing function the `WriteVoxel()` function is called for the currently set brush together with the location of the voxel in the world.

4.3.1 Drawing Functions

A number of standard 2D drawing functions were adapted to voxel drawing for this project. Functions were added as needed, with all of them using descriptions found in Computer Graphics: Principles and Practice by Foley, van Dam, Feiner & Hughes, as a basis. This section will outline each of the algorithms used and how they work within the voxel framework. Due to the fact that they are just simple modifications of well known and understood algorithms this will be brief.

4.3.1.1 Draw Line

For line drawing a simple adaptation of the standard Bresenham line drawing algorithm is used. This is fast and works perfectly for the purpose of this project. The Bresenham algorithm for drawing lines is attractive because it uses only integer arithmetic and does not require any rounding operations (Foley, van Dam, Feiner, & Hughes, 2001). Although the speed of rounding a few floating point values to integers is no longer the same concern that it was when the algorithm was developed it is still a very elegant method for drawing lines.

```
public static void DrawLine( int x0, int z0, int x1, int z1, int y )
{
    if( _brush == null )
    {
        throw new InvalidOperationException( "Set a brush before making a draw call!" );
    }

    // Adaptation of the Bresenham Line Drawing algorithm for 3d.

    int dx = Math.Abs( x1 - x0 ), sx = x0 < x1 ? 1 : -1;
    int dz = Math.Abs( z1 - z0 ), sz = z0 < z1 ? 1 : -1;
    int err = ( dx > dz ? dx : -dz ) / 2, e2;

    while( true )
    {
        _brush.WriteVoxel( new GlobalVoxelPos( x0, y, z0 ) );

        if( x0 == x1 && z0 == z1 )
            break;
        e2 = err;
        if( e2 > -dx )
        {
            err -= dz;
            x0 += sx;
        }
        if( e2 < dz )
        {
            err += dx;
            z0 += sz;
        }
    }
}
```

Note that in the code segment above, `_brush` is a variable of type `IBrush` that is a member of the `Drawing` class that I created. For more information on brushes and the `IBrush` interface that was created for this project see section 4.3.2 Brushes.

4.3.1.2 Draw/Fill Circle

For drawing circles we exploit the eight-way symmetry of circles, thus only needing to compute one 45° segment to produce a full circle (Foley, van Dam, Feiner, & Hughes, 2001). This is done with a procedure called `CirclePoints()`, the implementation of which is shown below:

```
private static void CirclePoints( int x, int y, int z, int xCenter, int zCenter )
{
    _brush.WriteVoxel( new GlobalVoxelPos( x + xCenter, y, z + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( z + xCenter, y, x + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( z + xCenter, y, -x + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( x + xCenter, y, -z + zCenter ) );

    _brush.WriteVoxel( new GlobalVoxelPos( -x + xCenter, y, -z + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( -z + xCenter, y, -x + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( -z + xCenter, y, x + zCenter ) );
    _brush.WriteVoxel( new GlobalVoxelPos( -x + xCenter, y, z + zCenter ) );
}
```

It then uses the midpoint circle scan-conversion algorithm to create the actual circles. This is also adapted from the algorithm presented in Foley, van Dam, et al, and originally developed by Bresenham.

```
public static void DrawCircle( int xCenter, int zCenter, int y, int radius )
{
    if (_brush == null)
    {
        throw new InvalidOperationException("Set a brush before making a draw call!");
    }

    int x = 0;
    int z = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints( x, y, z, xCenter, zCenter );

    while( z > x )
    {
        if( d < 0 )
           d += 2.0 * x + 3.0;
        else
        {
           d += 2.0 * ( x - z ) + 5.0;
           z--;
        }
        x++;
        CirclePoints( x, y, z, xCenter, zCenter );
    }
}
```

This algorithm draws a circle in the horizontal (x, z) plane, one voxel thick (with the default brush that simply outputs one voxel for every `WriteVoxel()` call; a different brush could in theory draw multiple voxels to create a thick outline instead).

A slightly modified version of `CirclePoints()` is used in the `FillCircle()` function. It fills in the spans between the edges of the circle, creating a filled circle.

4.3.1.3 Fill Polygon

For polygon drawing only a filling variant was created, since an outline drawing version was not needed. Like the circle drawing function this draws the polygon in the horizontal (x, z) plane of the world. The code used is presented below:

```
public static void FillConvexPolygon( Point2D[] points, int y )
{
    if( points == null )
        throw( new ArgumentNullException( "Points must not be null!" ) );
    if( points.Length < 3 )
        throw( new ArgumentOutOfRangeException( "Polygon must have >3 points."  ) );
    if( _brush == null )
        throw ( new InvalidOperationException("Set a brush before making a draw call") );

    Dictionary<int,MinMax> dict = new Dictionary<int,MinMax>();

    // Back up the current brush, we'll need to replace it once we are done building the
    // dictionary of polygon points:
    IBrush oldBrush = _brush;

    // Use a special type of brush to store all the polygon points:
    _brush = new PolygonBuilderBrush( dict );

    // Now we can just the regular line drawing function to build the dictionary of
    // points:
    for( int i = 0; i < points.Length; i++ )
    {
        if( i < ( points.Length - 1 ) )
            DrawLine( points[i].x, points[i].z, points[i+1].x, points[i+1].z, y );
        else
            DrawLine( points[i].x, points[i].z, points[0].x, points[0].z , y );
    }

    // Return the brush to the old value:
    _brush = oldBrush;

    foreach( KeyValuePair<int,MinMax> kvp in dict )
        DrawLine( kvp.Value.min, kvp.Key, kvp.Value.max, kvp.Key, y );
}
```

This particular implementation only works correctly with convex polygons. Those with concave areas will be drawn incorrectly. In practice this was not much of a problem since it is mostly used to draw convex polygons (although some castles can end up being generated with concave wall sections, causing small problems).

The algorithm uses the brush system (see section 4.3.2 Brushes) to create a list of all the edge pixels and then draws lines to fill the spans between the left and right edges of the polygon.

4.3.1.4 Fill Below

A special function was added to the drawing toolbox for filling in an area below a certain point. When placing the wall or towers sometimes canyons or exposed caves cut through the placement area. In order to prevent a wall from being left with a large gap underneath it when it spans one of those areas, or to prevent towers from being built with only half a foundation these areas are filled in first. When buildings are constructed they call this function first with the y-position of the lowest point that they will be built on. This function is called first, filling in empty blocks until it encounters a non-empty block, at which point it returns.

4.3.1.5 Clear Above

While the castle placement algorithm tries to avoid placing the castle such that it cuts into high terrain this still happens sometimes. To prevent terrain from blocking a walkway on the castle wall this function is called when a castle wall is created. It simply iterates through all voxels above the point given up to some maximum height value and sets them to be empty (i.e. "Air" blocks). This method is really a cheat to improve the look of the castle in situations where the placement is poor. Early in development it helped to clear the walkways when walls would be set into the mountain areas. As the placement algorithm improved this became a rare occurrence.

4.3.2 Brushes

This section covers the brush system used by the drawing code. It will explain why this approach was chosen and its benefits. An exhaustive list of all the brushes used in this project with explanations of what they do can be found in Appendix A.

Conceptually using a system of brushes is very simple. Instead of writing voxels to the world directly the drawing functions call `WriteVoxel()` on the currently set brush instead. All brushes implement the `IBrush` interface created for this project. It only has one function prototype: `void WriteVoxel(GlobalVoxelPos p)`. `GlobalVoxelPos` is a structure containing three integer values (x, y, and z), representing a single voxel location in the world.

The brush can then output a single voxel to the world, or it can do something else entirely. This makes it simple to implement all sorts of effects, such as outputting random blocks or creating a stipple pattern. Some, like the `FillBelowBrush`, output multiple voxels in the y-axis at the given x, z location. It is so flexible in fact, that a number of radial search functions used to calculate castle placement (such as the one used during initial placement covered in section 4.5.2.1 Initial Placement, or the one used to optimize tower placement in section 4.5.2.3 Optimization Step) are implemented by creating custom brushes and then simply using the `FillCircle()` drawing method to scan the required area (for more on this see section 4.5.1 Placing the Castle later on in this chapter).

4.4 Terrain Analyzer

In order to be able to successfully place the castle we need to first analyze the terrain so that we have enough information to make decisions on how and where to place the castle. To create the maps for this the brush system described in section 4.3.2 Brushes is used. A square that covers the entire created world is used in the `DrawPolygon()` function to create the maps, including images that can be saved to be examined later.

4.4.1 The Smooth Heightmap

This section is motivated primarily by the wall placement. This is discussed in more depth later in section 4.5.3.2 The Wall.

If we place the castle walls fitted to the terrain directly we experience problems even on the mostly flat terrain. Small rises and falls in the terrain are mirrored on the castle walls, creating a very unrealistic look (for an example see Figure 45 below), and if there are any places where the slope is greater than one block a break appears in the walkway on top of the wall (this can be seen in Figure 46 below).

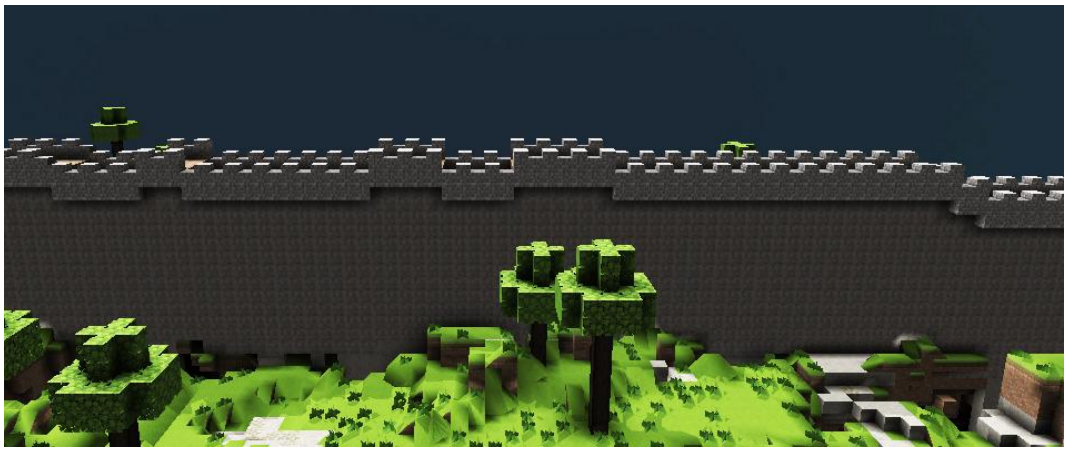


Figure 45 - Wall Following the Terrain

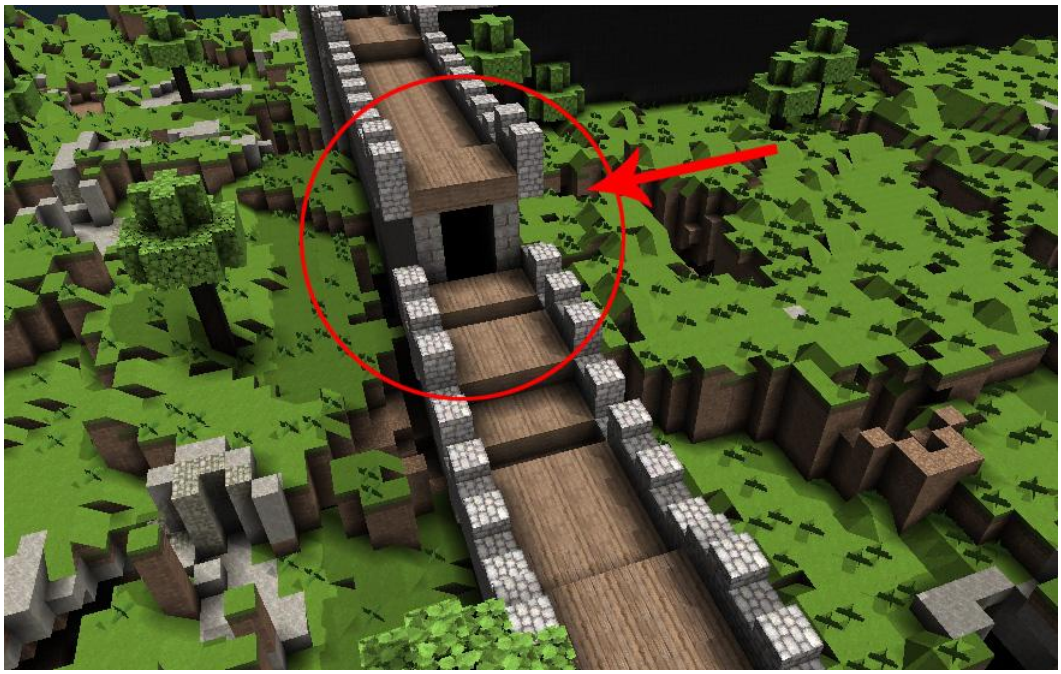


Figure 46 - Problematic Wall Section

In order to combat both the high frequency noise in the heightmap and remove large jumps up and down, a smooth version of the heightmap is generated. For this purpose a specialized blur function is used. The heightmap used for this purpose is the raw heightmap generated by the terrain generator and may not be accurate to the exact height in the final terrain. Specifically, it does not take into account any caves or canyons carved into the terrain by the ridged multifractal noise component in the terrain generator. This is done on purpose since we prefer walls to span any canyon areas anyway, rather than dip down to follow the terrain in those cases.

The blur function takes the heightmap as input and transforms it in three steps. First it scans across the heightmap and for every value checks the neighbouring four values (i.e. above, below, to the right and to the left), and then ensures that the current value differs at most by one from these.

Next the blurring is applied. A modified Gaussian blur was tried for this but the results proved unsatisfactory. Rather than a regular blur we actually want to create a series of plateaus. A function called `RadiusAverage()` was created for this purpose. The function takes a radius value and the heightmap as input. The heightmap is then divided into even squares of size of the radius passed into the function. An average value (i.e. the mean) is calculated for each square and values

inside the square are nudged towards this average (four passes are made across the values and any under the average have 1 added to them in each pass, while those over the average have 1 subtracted from them).

The `RadiusAverage()` function is called four times across the heightmap data, with the output fed into the next call of the function each time. It is called with a radius value of 64 first, then 32, 16, and lastly 8. This ensures that large areas are averaged out first and then smaller areas. The values were chosen to ensure a relatively smooth outcome while still maintaining something of the underlying terrain structure.

Once the blurring is complete a final pass is done to ensure that after all this there is still no difference in neighbouring values greater than 1. This is simply a repeat of the first step in the process.

The result can be seen in Figure 47 below, showing a heightmap before and after the smoothing operation has been applied.

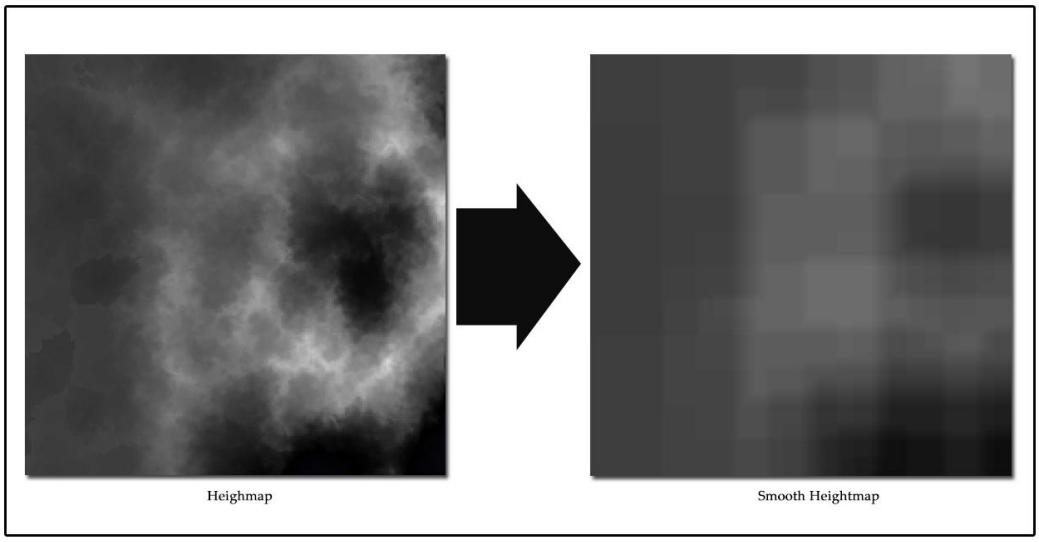


Figure 47 – Heightmap and Smooth Heightmap

Placing castle walls using the smooth heightmap as a basis isn't perfect, but it is a huge improvement on placing them on the terrain directly. Small adjustments were added to the wall placement function to iron out any remaining issues. These are discussed in section 4.5.3.2 The Wall.

4.4.2 Calculating the Terrain Fitness Score

Placing the castle (explained in section 4.5.1 Placing the Castle) using the smooth heightmap generated as described above resulted in much better looking sections of wall but resulted in another problem. Because the heightmap being used no longer matched the underlying terrain as closely it meant that in places the wall would be very low. Imagine a steep hill for example, the smooth heightmap using an average value of the area and never moving up at more than one unit per voxel, will have values much lower than the real terrain. When the wall is placed with a fixed height value above the height calculated from the smooth heightmap it now might not even reach the real terrain height (if the difference between the smooth heightmap and the real heightmap is greater than the chosen wall height). This is illustrated in Figure 48 where we can see a wall section set right into a hill. Attackers could simply walk up the hill and step onto the walkway, negating the entire point of having a wall at all.

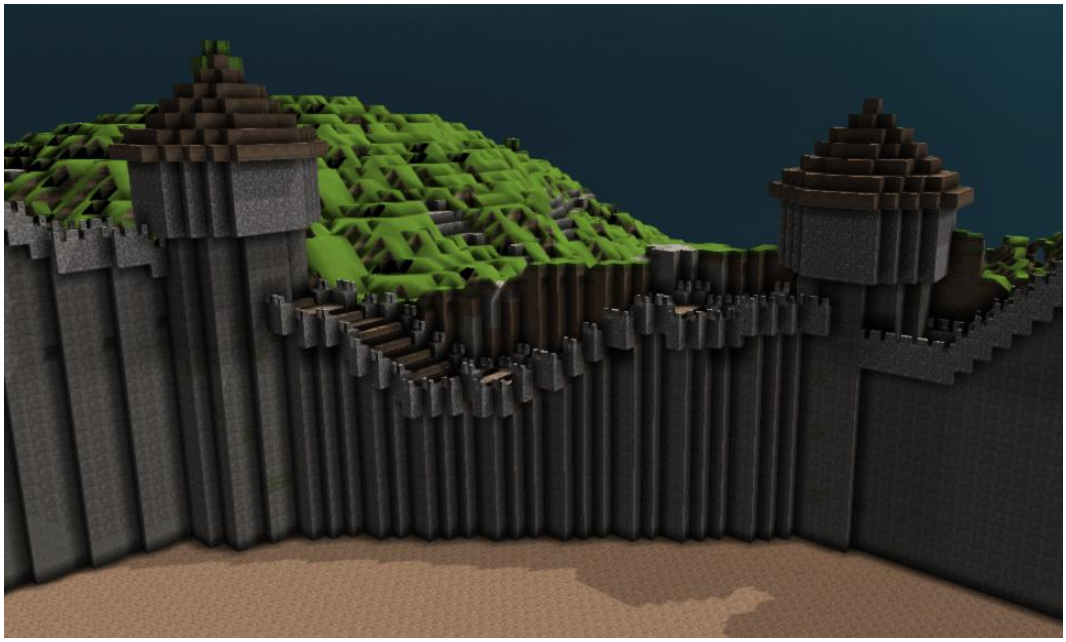


Figure 48 – Another Problematic Wall Section

In addition to avoiding steep hills we might also wish to avoid steep canyons, since they will require substantial foundation filling work, lest someone be able to access our castle by simply walking along a canyon or ravine and getting under the wall. This would be unrealistic and would result in a castle with poor defensibility (for example making it easy to undermine the wall, something discussed in the literature review in section 2.1 Medieval Castle Construction).

Due to how the terrain generation works (see section 4.2.7 C) the heightmap might not be entirely accurate. In terrain generation the heightmap is only the starting point, with caves subtracted from the terrain in a separate step, and where those caves intersect the terrain they can create canyons and ravines. So for this purpose a "real" heightmap of the terrain is generated by scanning the entire terrain from the top down until a non-empty block is encountered, and the height of that block is recorded. This gives us the true height values at each point, including trees, and any areas where the cave system has added deep cuts into the terrain.

The result of this step can be seen in Figure 49 below. Note the deep black areas, particularly visible in the top left of the image. This is where the cave system has penetrated the surface and created deep cuts into the terrain. These are areas that should be avoided by the castle.

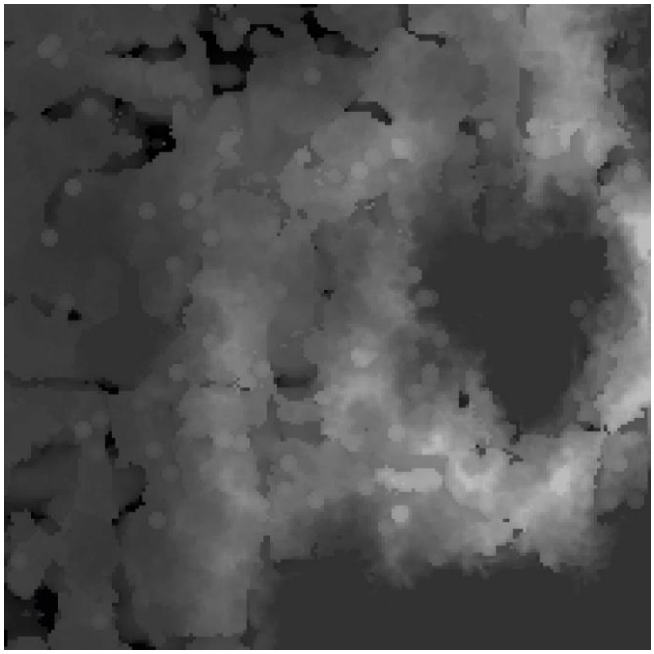


Figure 49 – Real Heightmap

For the next step the absolute difference is taken between the smooth heightmap and the real height values. The negative of this value is then recorded as the terrain fitness score for that location. This is visualized in Figure 50 below. Black regions are close or equal to zero, red indicates negative values and white areas are very high negative values.

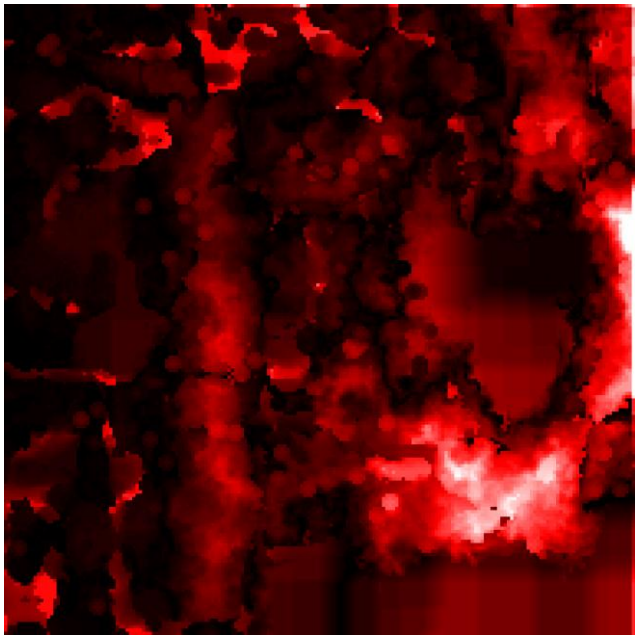


Figure 50 – Terrain Fitness Map

With the terrain fitness score for every location in the horizontal plane calculated, this can now be used to make decisions on where to place the castle. Avoiding placement on areas with high negative values will prevent the castle from being built in a way that creates poorly defensible walls (such as the one seen earlier in Figure 48). This value, the terrain fitness score, is the heart of the castle placement algorithm used in this project and represents the key innovation developed. Together with the smooth heightmap it could be used to place any sort of man-made structure into a voxel world and ensure at least somewhat plausible placement. Note in Figure 50 how the black areas identified as deep cuts into the terrain by the cave system in Figure 49 are now bright red, meaning that they will be avoided by the castle placement and layout algorithm (this is explained in more detail in section 4.5.1 Placing the Castle).

4.5 Castles

Castle construction is a multipart process. There are a number of parameters available for castle construction that can be adjusted to get the desired result. These are passed into the castle constructor, which uses them to first place the castle, then adjust its shape to better fit the terrain, and finally constructs the actual castle by placing voxels into the world using the voxel drawing system created for this project (covered previously in section 4.3 Drawing with Voxels).

The castle placement and layout optimization process is mostly heuristic based. It functions mainly by calculating an overall fitness score for the castle placement, and then performing an exhaustive search across the terrain to find the best placement. The aim is to create an overall algorithm that will place the castles correctly in a wide range of terrains. By “place correctly” the following criteria were identified as being important: firstly the castle layout should avoid problem areas such as tall mountains or deep gorges, which would cause significant problems during the actual castle construction (such as walls or towers set deep into a mountain). Secondly, the algorithm should prefer to place the castle in defensively advantages positions. That is, the castle should be preferred to be placed on high ground where possible. While there are many other criteria that a real castle builder might have to contend with when choosing a site for building (such as ease of access, nearby resources, access to clean water), the two chosen criteria were considered the most important to create a castle that will look plausible to players of a voxel world game. They are also universal to all types of terrain that a game might generate, while, for example, consideration for nearby resources would be much more game specific (it would depend on the types of resources the game has, as well as which of these might be important to a player owning a castle).

In theory parts of the castle construction process could be randomized, but for this project a purely deterministic approach was used instead. That is, given a particular terrain to work with the castle generation process will create the exact same castle every time. Variation in castle layouts is driven entirely by variation in the terrains that the castles are placed onto. Chapter 5 will demonstrate a number of castles generated and show that the algorithm is nevertheless capable of significant variation.

This section covers the parameters and how they affect castle construction, and details the process of finding an initial placement for the castle, how the castle shape is adjusted for the terrain, and finally how it is actually built into the voxel world.

4.5.1 Placing the Castle

Castle placement is done in several steps. The first is to find an overall starting position. Once a position has been found the castle towers are moved around to find the best fit. The structure of the castle can be thought of as a polygon, with the towers making up the vertices and the walls the edges (Figure 51 shows what this basic configuration looks like for a castle with 8 towers). The shape of this polygon is adjusted such that it fits well into the given terrain.

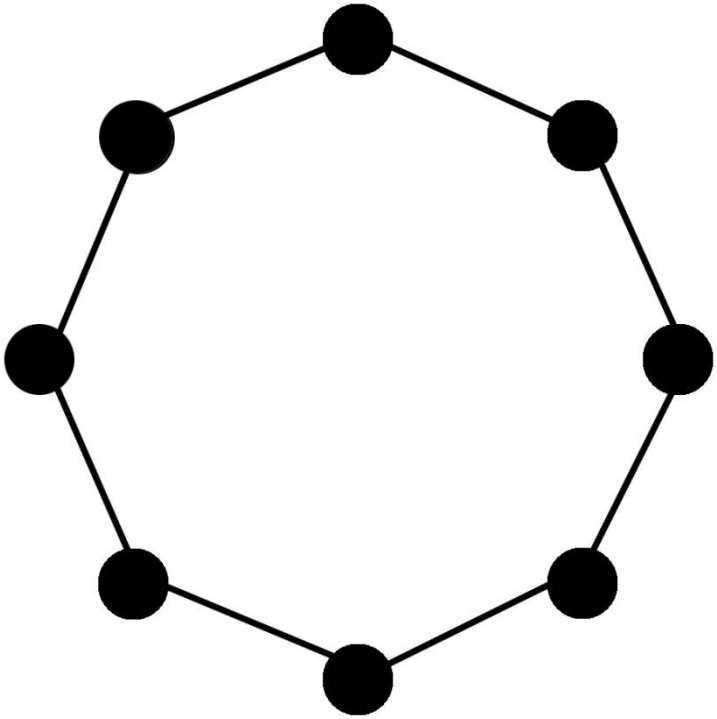


Figure 51 – Basic Castle Layout Shape (8 Towers)

The castle placement and layout optimization step does not involve placing any voxels in the world. It is a search-and-optimize algorithm using the terrain fitness map (see 4.4.2 Calculating the Terrain Fitness Score) and the heightmap (see 4.2.6 Creating the Heightmap).

4.5.2 Base Parameters

Castles in VoxBox can be constructed with a number of parameters that are listed and explained below:

- `Position` (`Point2D`): The position of the centre of the castle in the world, given as an x and z value.

- `Width` (`int`): The diameter of the castle to be generated. The name is a holdover from early versions where the size of the castle was described in terms of a width and depth value to create a rectangular castle.
- `NumberOfTowers` (`int`): Number of tower nodes to generate for the castle (towers in the outer wall of the castle). This value must be 3 at a minimum (smallest number of points a polygon can have and have a non-zero area). The value given here is used as a starting value and there is no guarantee that this is the number of towers the final castle will have due to the fact that close towers are merged during the final step of castle generation. To start with towers are laid out evenly spaced along a circle with a diameter equal to the width value mentioned above.
- `OptIterations` (`int`): Number of iterations over the tower placement optimization algorithm (described in section 4.5.2.3). The thought was that multiple iterations over the optimization function might achieve better results. However in practice this resulted in problematic castle layouts, shrinking the castle area too much or creating overlapping walls. For this reason the value is best left at 1.
- `InitialPlaceSearchRadius` (`int`): The radius of the initial castle placement search step. For more information see 4.5.2.1 Initial Placement.
- `PreOptSearchRadius` (`int`): Radius to search for the pre-optimization step of tower placement.
- `OptSearchRadius` (`int`): Radius to search when optimizing the tower placement. During testing it was found that it is best if this value is smaller than the one given for `PreOptSearchRadius`.
- `BidirectionalSearch` (`bool`): Defines if the tower placement optimization step should use a bidirectional search algorithm. This is explained fully in section 4.5.2.3 Optimization Step. During testing this was not found to be very effective however, and generally left as false.
- `TowerMergeRadius` (`int`): If the distance between towers is found to be smaller than this value they are merged into one. This is to prevent towers from ending up very close to one another or even intersecting.

See Appendix B for the values that were eventually chosen to create the majority of the example castles shown in this document.

4.5.2.1 Initial Placement

For initial placement of the castle we wish to avoid areas that have particularly poor terrain fitness scores (such as the middle of a mountain range). To this end a test configuration of towers is calculated for the castle that is being placed. This configuration is tested at each location across the search radius to find the best position.

For each test castle configuration an overall fitness value is calculated as:

$$\left(\sum_{i=0}^{n} heightAt(x_i, z_i)\right) - \left(\sum_{i=0}^{n} terrainFitnessAt(x_i, z_i)\right)^2$$

Where n is the number of towers and the coordinates (x, z) are the locations of each of the towers in the castle configuration. What this calculation gives us is a value that scales linearly with the sum of the terrain height and quadratically with the sum of the terrain fitness score. This means it will favour high areas on hills over valleys but will avoid steep mountains.

The castle configuration is then rotated slightly the same process is repeated. This is done a total of five times. The angle of rotation (in degrees) is calculated as (n is again the number of towers for this castle configuration):

$$\frac{\left(\frac{360}{n}\right)}{5}$$

If the calculated score is better than the previously best calculated score then this score becomes the new best score and the position as well as the rotation offset is recorded.

To evaluate the terrain for a suitable position the brush system is utilized. A new brush, the `InitialTowerPlacementOptimizerBrush`, was created that creates a test castle configuration at each point that `WriteVoxel()` is called for (using just the x and z coordinates passed in, the y value is discarded).

This brush is attached to the drawing context and the circle fill function is used with a radius equal to the `InitialPlaceSearchRadius` value passed into the castle generator. To slightly speed up this operation only every second position is evaluated by the brush. In the case of an actual game, where speed is important, it would probably yield acceptable results testing more sparsely than that. Perhaps even every eighth position would be fine since this is only the initial placement. Doing away with trying different rotation offsets would also bring a 5x speed increase and in most cases would probably not make a large difference. For testing, with a search radius of 140 and a castle configuration with 10 towers a total of 154,975 different positions were tried in ~15 seconds (on an i7-4712HQ).

4.5.2.2 Pre-Optimization Step

In the pre-optimization step individual tower positions are adjusted for better placement. A custom brush is used, the `PreOptTowerPlacementBrush`, and it makes use of the same formula as the initial placement. For each tower a circular area with a radius equal to the `PreOptSearchRadius` is analyzed. At each point a value v is calculated like this:

$$v = height - (terrain\ fitness\ score)^2$$

The point with the highest value is chosen and the tower is moved to that position. Once again, this value scales linearly with terrain height and exponentially with the terrain fitness score. This ensures that towers are placed in high areas where possible but avoid steep terrain features. Because the tower positions are moved again by the next step in the process it is preferable if the radius for the pre-optimization step is larger than the radius for the next step. Otherwise, the final position will be dominated almost solely by the final optimization step.

4.5.2.3 Optimization Step

So far castle placement has focused only evaluating the terrain at the points that the towers will be placed. This step instead looks at what the terrain is like between the towers (i.e. where the wall will be built). Consider for example a scenario where there is a steep hill between two towers like shown in Figure 52. If possible we want to move the towers so that they avoid the obstruction and the wall is able to take a clearer path.

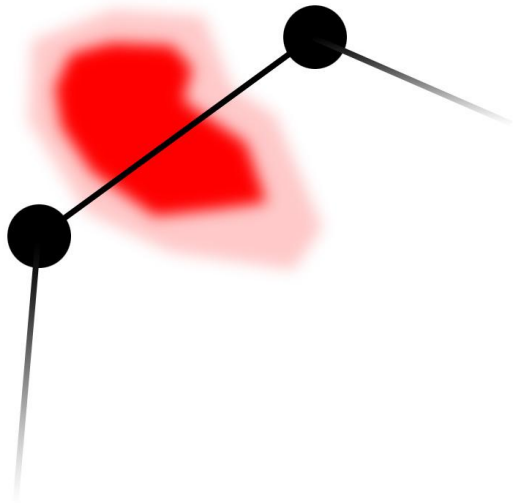


Figure 52 - Obstruction between Towers

For each tower a search-and-optimize step is done where the tower is moved around in an area surrounding the tower within the radius defined by the `OptSearchRadius` parameter. At each position the worst terrain fitness score that is found between this tower and the next is recorded. If bidirectional search is set to true then the same is done between this tower and the previous tower. At the end the position that yielded the best terrain fitness score is chosen as the new location for this tower.

A limitation of this approach is that only one tower is moved at any given time. This means that the algorithm may be incapable of finding the best solution possible in some cases. Consider the case shown in Figure 53. The red area is a hill we wish to avoid and the black circles are where the towers are now. The grey circles show the optimal placement of the towers so that the wall (the line between the tower circles) avoids the worst of the hill. This configuration cannot be found by only moving one of the towers at a time and optimizing for the best terrain fitness between the towers.

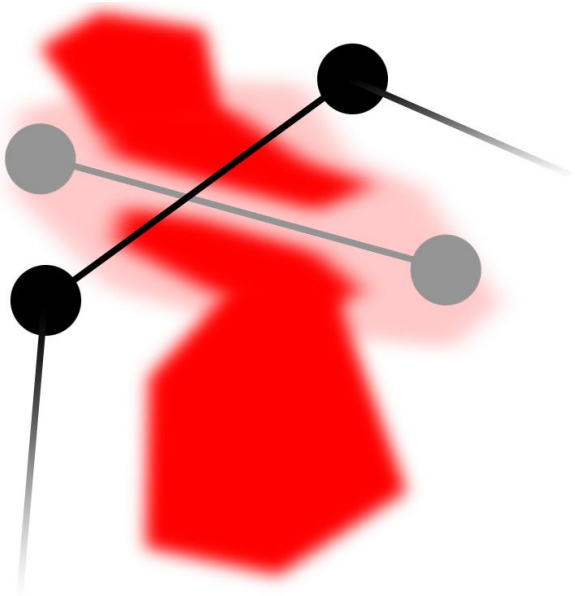


Figure 53 - Difficult Obstruction between Towers

An attempt was made to improve this by iterating over the optimization step. This produced some interesting results but proved ultimately unfruitful. It did not create noticeably better results; in fact the more iterations were tried the worse the final castle layouts became (see Figure 54 showing visualizations of castle layouts in green on their terrain fitness map).

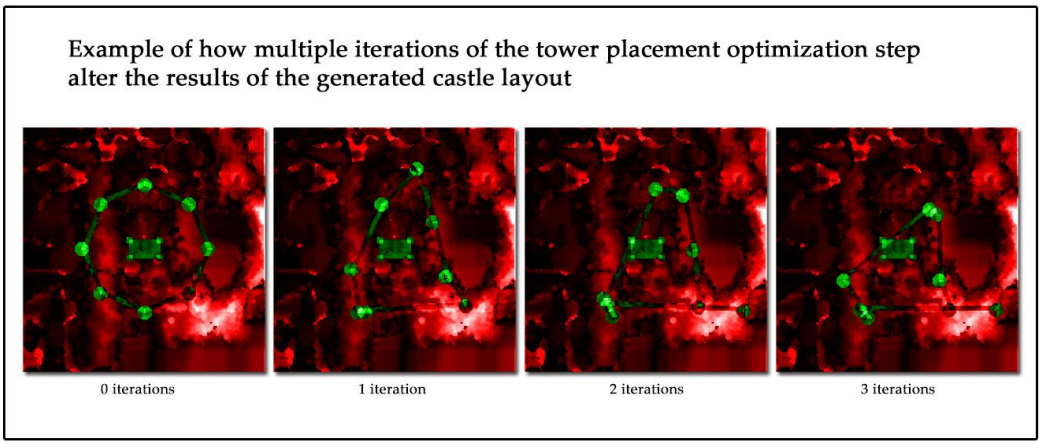


Figure 54 - Iterations over the Optimization Step

Because every pass of the optimization step is able to move the towers further the castle layout becomes more disconnected from the starting configuration. Without any additional constraints this means that often the castle begins to shrink, walls start to cross paths and many towers end up close enough together to be merged (see 4.5.2.4 Merge Close Towers Step). For this reason it was decided to just perform a single pass of this step for the examples shown in this thesis.

4.5.2.4 Merge Close Towers Step

If the initial distance between towers is large enough (either because of a low number of towers, or a large build radius), or the search radii for the two tower placement optimization steps is kept small enough, then this step would be unnecessary. However, during testing it was found that better results were achieved by the algorithm when a tower density and search radius size were used that could on occasion result in closely placed towers. This did mean however, that sometimes towers would be placed only a few blocks apart or would even overlap (an example of this can be seen in Figure 55 below). For this reason a final check was added to the castle placement calculation to detect towers within a certain distance of each other, which could be merged into single towers.

Figure 55 - Two Towers Generated Close Together

For this step the list that stores the tower points is traversed and a new list is built. For each node the distance to the next node is checked and if it is greater than the threshold value then it is added to the output list, otherwise it is discarded. Note that only neighbouring towers are checked. In a particularly pathological configuration it may be possible for non-adjacent towers to become placed close to each other, and this would not be detected. This becomes more likely when large search radii are utilized for the tower placement and/or the optimization step is iterated two or more times. This problem never manifested with the test cases that were run (with small search radii, and only one iteration of the optimization

step). Since merging non-adjacent towers is not possible, should this be found to be an issue another solution may need to be found to move the towers in such situations so that they are not too close.

One other benefit of this step is that it naturally adds additional variation to the castles produced by changing the number of towers generated for the outer wall. This could, of course, also be achieved by feeding a random value for the number of towers to produce into the castle generator, and indeed this is still an option if greater variation is desired. If a specific number of towers is desired for some reason and the uncertainty introduced by this step is unwanted it can easily be disabled by setting `TowerMergeRadius` to zero.

4.5.3 Constructing the Castle

For the purposes of this project only a simple castle is constructed: a number of towers connected by walls and a keep in the middle of the area enclosed by the outer wall. The focus was figuring out how to place the castle sensibly within a randomly generated world and how to adapt the overall shape of the castle to the terrain. Creating a more detailed keep and additional buildings such as stables, an armoury and/or a chapel is left as an open exercise.

Castle construction itself is done with a modular approach. Functions were created to create different types of towers, and the brush system was used to create brushes for flattening the interior area of the castle, as well as to construct the outer castle walls.

4.5.3.1 Challenges

At this stage in the castle construction the basic layout and placement of the castle is already known, so all that remains is the actual placement of voxels into the world to create the castle. However, one major difficulty remains: at any given point in the castle we do not want to introduce any sudden changes in elevation leading to breaks in the wall. For this reason we use the smoothed heightmap computed previously (explained in section 4.4.1 The Smooth Heightmap). This ensures no large changes in elevation and prevents high frequency noise in the terrain height from breaking up the flow of the walls. It also means that the wall might be set into the sides of hills or mountains in some cases. While the castle placement algorithm specifically tries to avoid areas that would cause such

problems it is not always possible. So castle creation must deal with these situations as gracefully as possible.

Another particular challenge was dealing with the inside area of the castle (the courtyard). Initial tests tried to retain something of the underlying terrain by removing blocks down to the smooth heightmap but this was abandoned for a completely flat look instead (for more detail on this see section 4.5.3.3 The Inside/Courtyard).

Building the actual castle required placing a lot of voxels, a task made much simpler by making use of the voxel drawing system that was implemented for this project (see section 4.3 Drawing with Voxels for more details). For example: before the drawing system was added the initial tower creation code explicitly laid out where each voxel would be placed for each horizontal slice of the tower, requiring a lot of code and giving no flexibility to change the tower radius. With the voxel drawing system implemented this was changed to simply calling `DrawCircle()` at each slice and providing a radius, thus creating the tower cylindrical shape by building up multiple circles atop each other.

4.5.3.2 The Wall

Wall creation was again implemented with a custom brush, making it easy to draw the walls in the world simply using the `DrawLine()` function in my voxel drawing toolkit. Each call to `WriteVoxel()` in this case produces a one voxel section of wall. That is, it creates the entire piece of wall from the ground up and to a thickness and height specified when constructing the wall brush. The wall brush ran into some problems with the context-free nature of the brush system. Each call to the `WriteVoxel()` function of the brush is independent and without information on what shape it is drawing. However, the wall brush must have access to the direction the wall is being drawn in to be able to select the correct orientation to create the wall in. To make this information available the brush is given the length of the entire wall line that is expected to be drawn in both the x and z directions. By comparing the values a choice is made as to how the wall section will be drawn.

The wall is created by using the smooth heightmap (see section 4.4.1 above) as a starting point and creating the wall up from there to a specified height. This

ensures that the walkway on the top of the wall stays mostly even and does not rise by more than one voxel at a time. This is not perfect however, and in particular there were rare cases where a wall moving at an angle diagonal to the voxel grid would produce jumps greater than one voxel, resulting in a discontinuity along the top of the wall as can be seen in the figure below (Figure 56).

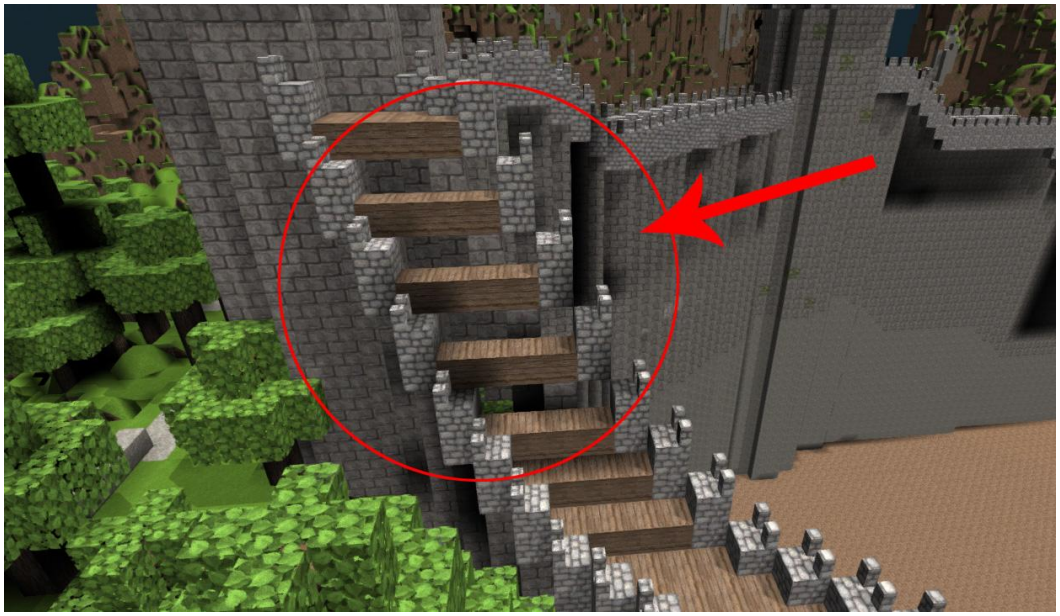


Figure 56 - Problematic Stairs

While this occurred only in a very small number of generated castles, when it did occur it was very noticeable. For this reason another check was added to the wall construction code to ensure that steps between parts of the wall would never change by more than one voxel in the y-direction.

4.5.3.3 The Inside/Courtyard

Two different approaches were attempted for constructing the inside space of the castle area. The initial approach was to shape the area to adapt to the smooth heightmap. This produced a result like this:

Figure 57 – Castle with Courtyard Adapted to Smooth Heightmap

The idea was that the rectangular areas created by the smooth heightmap would be perfect for placing various buildings and the inside of the castle would keep something of the underlying terrain.

In practice the results were usually not very visually pleasing. Perhaps with a full complement of buildings to cover the area it might actually look good but with just the keep placed in the interior area it simply looked wrong. It would also cause problems with the construction of the keep in some cases, causing it to float partly in the air and other issues.

So a more simple approach was used: the interior was flattened to a single elevation value resulting in something looking like Figure 58.

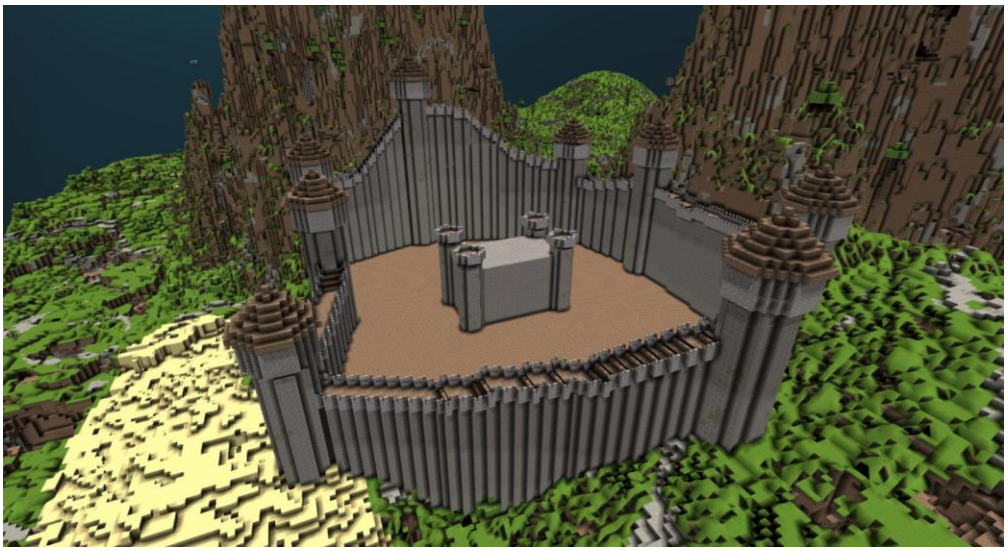


Figure 58 – Castle with Flat Courtyard

As for the elevation value used, this was first set to the lowest value found in the smooth heightmap across the surface of the castle area, however this caused the interior to be too low in some cases. This was changed so that the average height of the smooth heightmap across the castle area is used instead. This produces a good result in most cases, although it can result the castle interior area being higher than the wall in some places. This happens when one side has a very low wall and the other sides are very high. To prevent this, the interior height could be capped to be just below the lowest section of wall. This was not implemented in this project because the number of problem cases is very low (only 1 in 100 randomly generated castles exhibited this problematic behaviour) but would be trivial to fix should the algorithm be used in an actual game.

4.5.3.4 The Keep/Donjon

While the keep or donjon is an important feature of most castles it is only represented by a very simple facsimile for the purposes of this project. The focus of the project was placing the castle and adjusting the outer walls to the terrain, with everything else being secondary. Creating an interesting interior to the castle, including a more complex keep is something for another project (and indeed computing a sensible layout for the castle interior could be a whole project on its own).

Placement for the keep is computed simply by taking an average of all the outer wall tower positions. This produces the midpoint of the castle and the keep is constructed there. As long as the castle construction parameters are carefully chosen this produces a good result. If the tower search radii are made too large or if multiple iterations of the optimization step are used then the resulting castle layout may end up with the keep intersecting the outer walls, towers, or even outside of the castle interior area altogether. A more sophisticated keep placement algorithm might adapt to such layouts better, perhaps even rotating the keep to fit the interior area more optimally, and/or modifying the shape of the keep itself. However, within the parameters chosen for castle construction for this project the simple midpoint placement proved sufficient in all tested cases.

4.5.3.5 Moats

A common feature found on many real world castles is the moat. A moat provides an additional layer of defence for the castle and makes it more difficult to approach and breach the walls.

The addition of a moat was explored in this project and the functionality exists in the castle construction algorithm. This is done by constructing a ring around the castle slightly offset from the walls and digging down to slightly below sea level (set at 0). If water blocks are encountered during digging, a call is made to the `SpreadWater()` function in the terrain generator once the moat is complete. This floods the moat with water; otherwise it is left as a dry-moat.

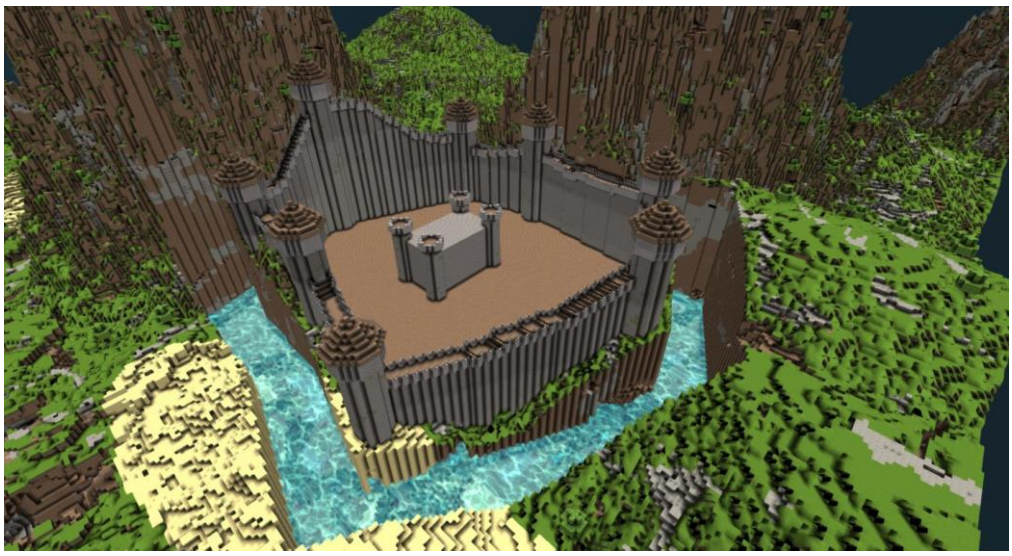


Figure 59 – Castle with Moat

The moat building functionality was not used during most of the testing, simply because it tends to hide any problems with the castle placement algorithm, by making a plausible looking castle almost anywhere. This is because it will demolish an entire mountain to create the moat if necessary. Perhaps a better approach would be to calculate the number of blocks that would need to be excavated in order to construct the moat and only build it if the number is below a certain threshold that is deemed appropriate. However, iterating over all the blocks that may be needed to create the moat, and then doing it again to actually construct it would be computationally expensive. A cheaper option may be to use the values from the heightmap to approximate a cost instead.

In the end not much more was done to explore the moat idea because it was orthogonal to the core of this project: the placement and adaptation of the castle on randomly generated terrain.

4.5.3.6 Future Expandability

As mentioned in the previous sections there are a number of possibilities for future work. With the castle placed, and the outer walls adapted to the terrain there are nearly endless possibilities for expanding the actual castle construction step to produce better and more varied castles. The first thing would be to create additional buildings to be placed in the interior of the castle. Additionally a section of wall could be designated as a gate and a gateway with guard towers (and perhaps a drawbridge if a moat is present) could be built there.

Another possibility for future addition is changing the blocks used in construction based on the biome where the castle is constructed (using different types of wood depending on the types of trees growing near the castle, using sandstone instead of regular stone for the walls in desert areas). Other adaptations could also be made such as using oriental-style towers when building in a desert biome.

Chapter 5: Demonstrations

In this chapter I will present a number of different terrains that were randomly generated and show how a castle was created to fit the terrain. I will point out areas where the algorithm performed well, and areas where it did not. As well as screenshots of the scenes, this section will make use of the terrain fitness maps as described in section 4.4.2 Calculating the Terrain Fitness Score and 4.5.2.3 Optimization Step. These show the terrain fitness score calculated for each point (worse scores in red, better scores in black) as well where the castle was constructed (in green). These make it possible to see how the castle placement algorithm perceives the world, and how well it did with its placement given this view.

The castle placements and layouts created by the algorithm will be examined against the criteria set for the algorithm (i.e. to maximize the height of the castle placement while avoiding areas with a poor terrain fitness score that would create problems during the castle construction). The 5 examples examined in this section were chosen from a set of about 30 terrain seeds that were used during the testing of the algorithm. They were chosen either because they produced an interesting result or because the terrain presented an interesting challenge. These examples also show the range of castles that are possible to be generated by the castle generation algorithm, simply through the variation in the terrain that they are placed in (recall that the castle generation algorithm itself is deterministic as discussed in section 4.5 Castles).

5.1 Example 1

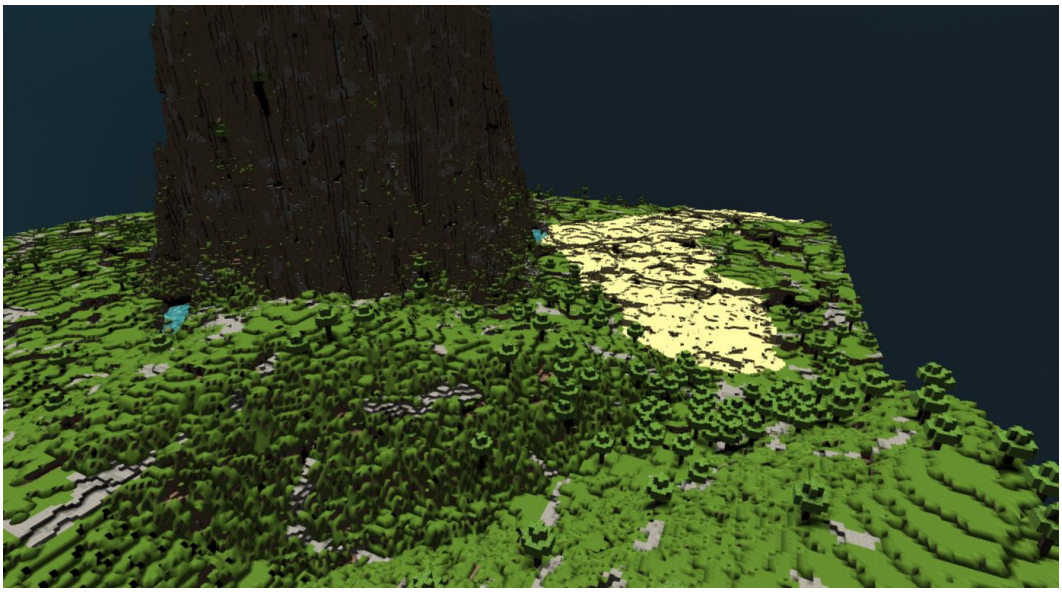


Figure 60 – Example 1 Terrain

As can be seen in the image above the terrain is generated with a large mountain area, surrounded by hilly areas.

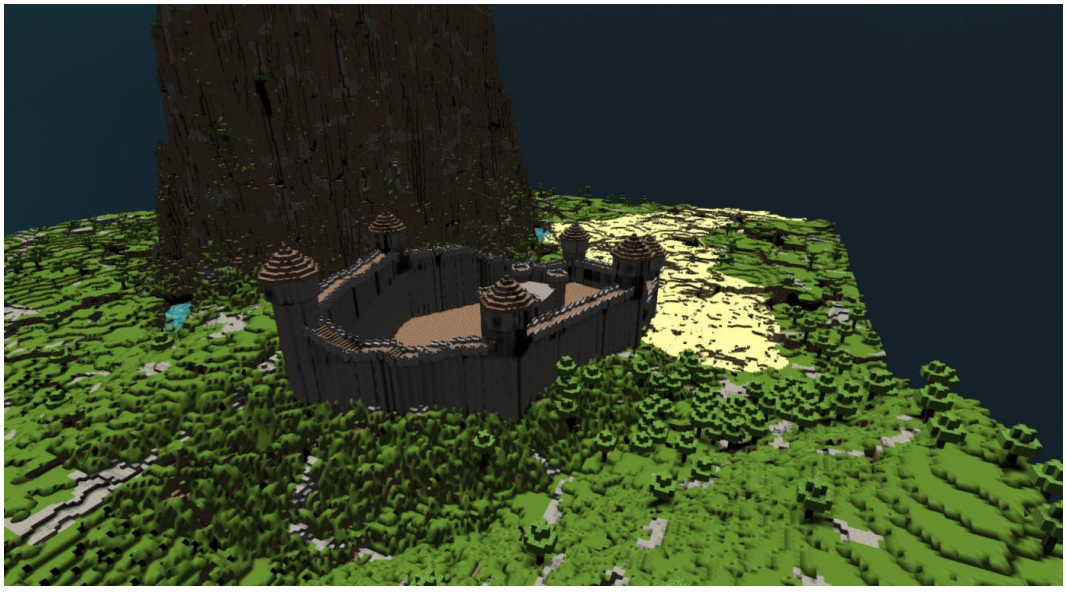


Figure 61 – Example 1 Castle

When the castle is placed into the terrain the placement selected is on top of one of the hills, creating a good defensive position, while avoiding the mountain. This is exactly the sort of result expected from the castle placement algorithm.

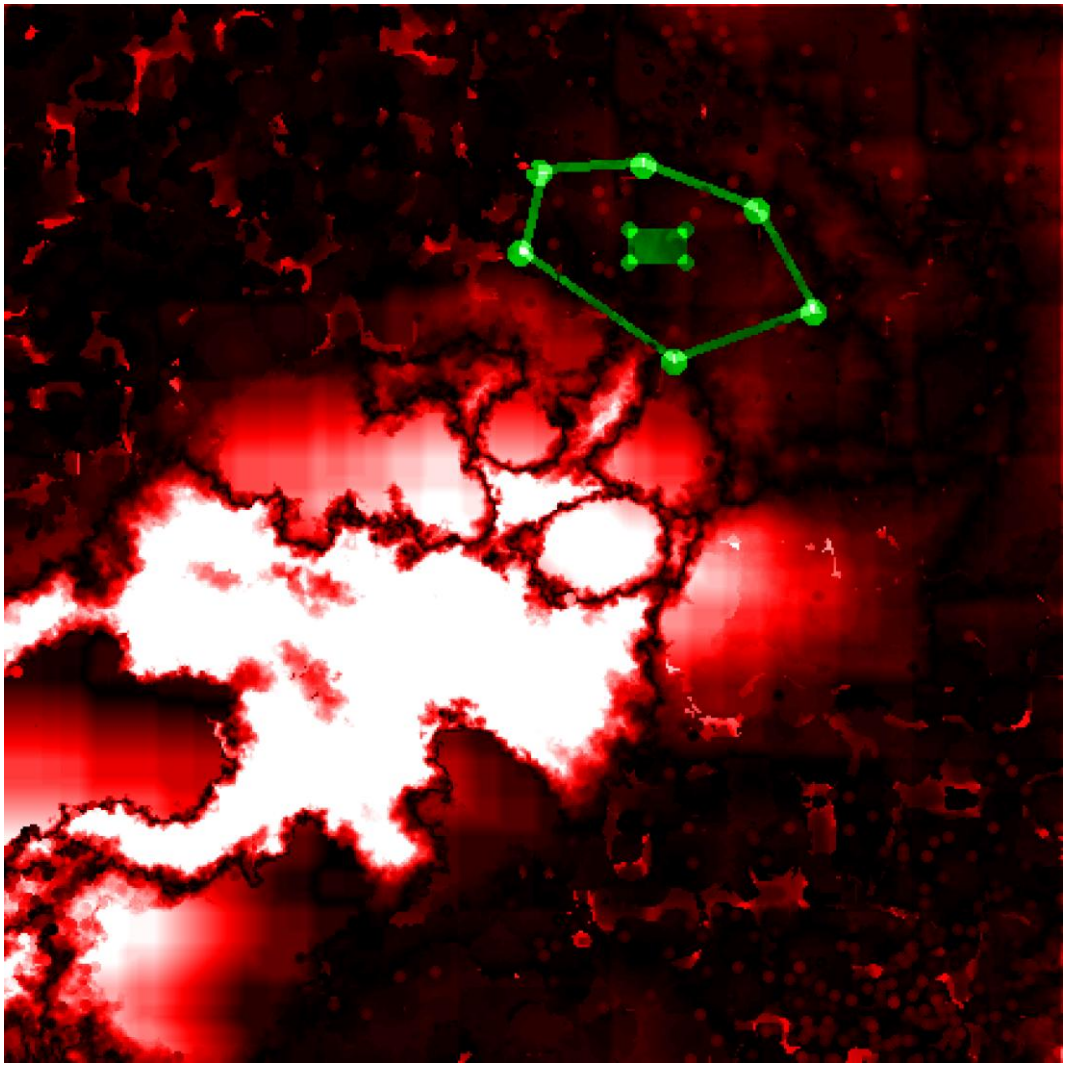


Figure 62 – Example 1 Terrain Fitness Map

Shown above is the terrain fitness map for this terrain, with the castle shown in green. As can be seen the mountain area is lit up in bright white, telling the algorithm to avoid this area. The bottom most two tower points sit right up against the area where it becomes more bright red (i.e. the terrain fitness score is lower), and the wall section between them is also right up against this area, without touching it. This can be deemed a satisfactory result then, according the criteria set for the algorithm.

5.2 Example 2

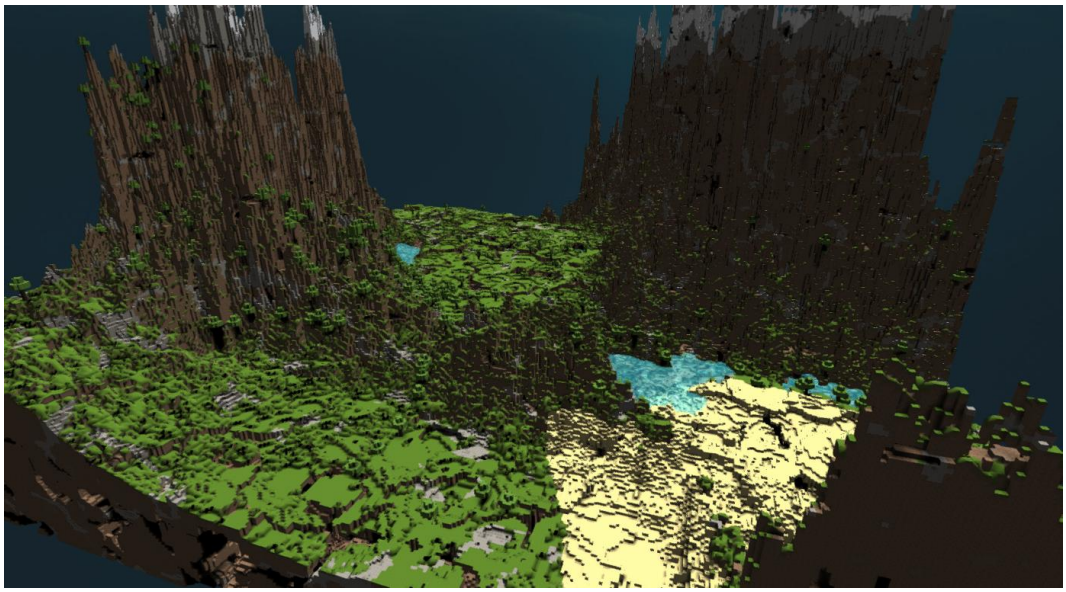


Figure 63 – Example 2 Terrain

The above terrain (Figure 63) presents an interesting problem. Two mountains, one on the left and one on the right, frame the terrain. In the middle there is a rough hilly area, flanked by a lake or estuary. Ideally, we would want to build the castle in one of the relatively flat areas near the front or back of the terrain.

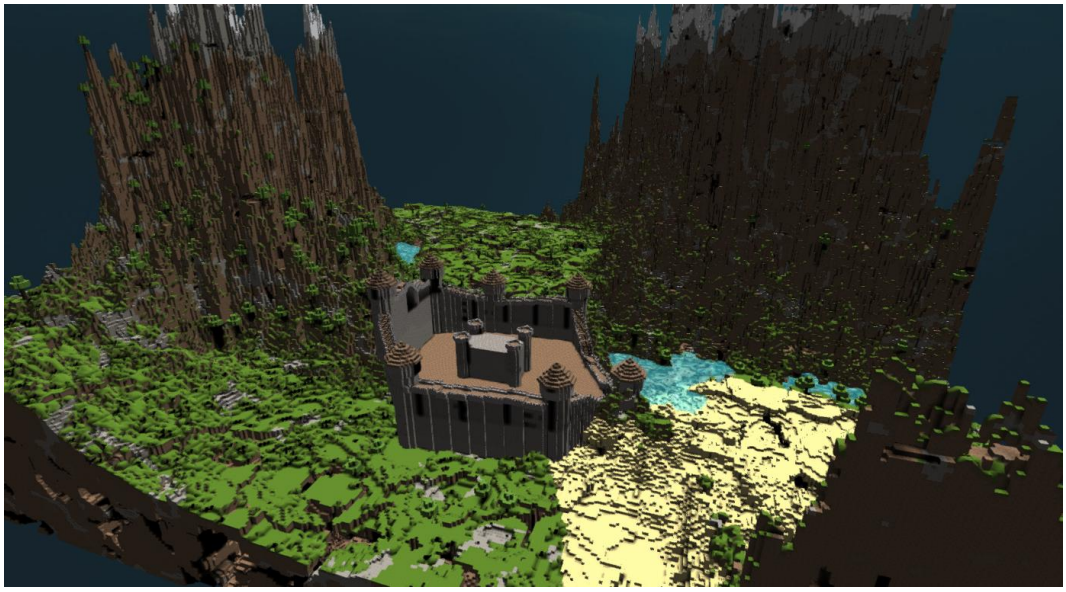


Figure 64 – Example 2 Castle

Instead the algorithm chose to place the castle near the middle on the terrain as can be seen in the image above (Figure 64). This is due to the hilly area providing some elevation, attracting the initial placement towards it. However, most of the hill is bulldozed during the castle construction anyway and one of the towers (the

rightmost tower in the image) is left very low because it is built partially into the body of water.

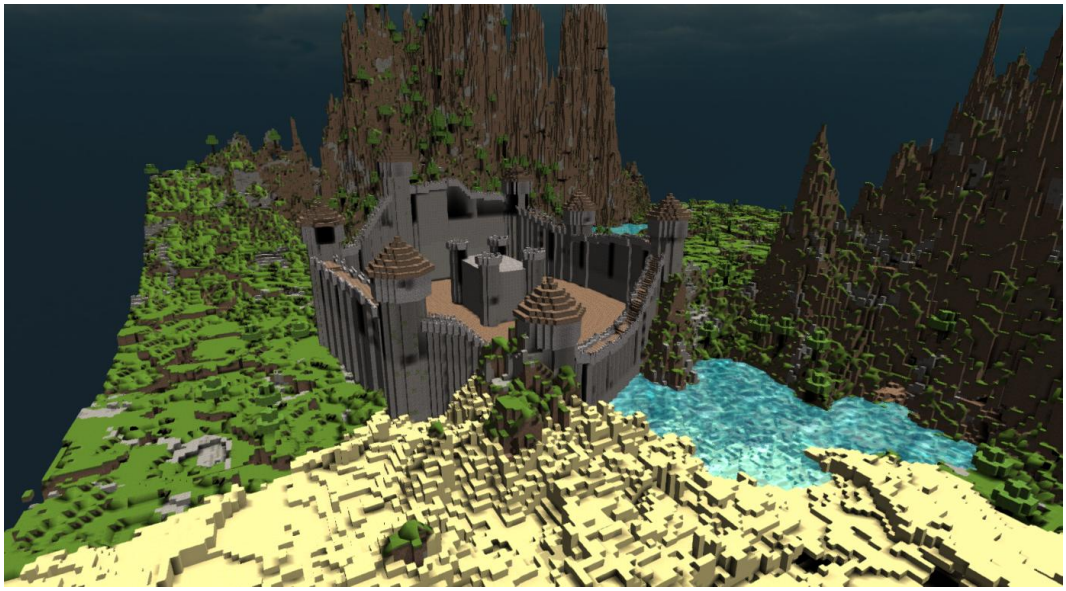


Figure 65 – Example 2 Castle from another Angle

Above is another screenshot (Figure 65) showing the problematic tower more clearly. It is pulled down because the smooth heightmap is influenced strongly by the body of water, and the very low height values in that area. This pulls the tower down with it and causes it to be embedded into the hilly terrain, rather than rising above it.

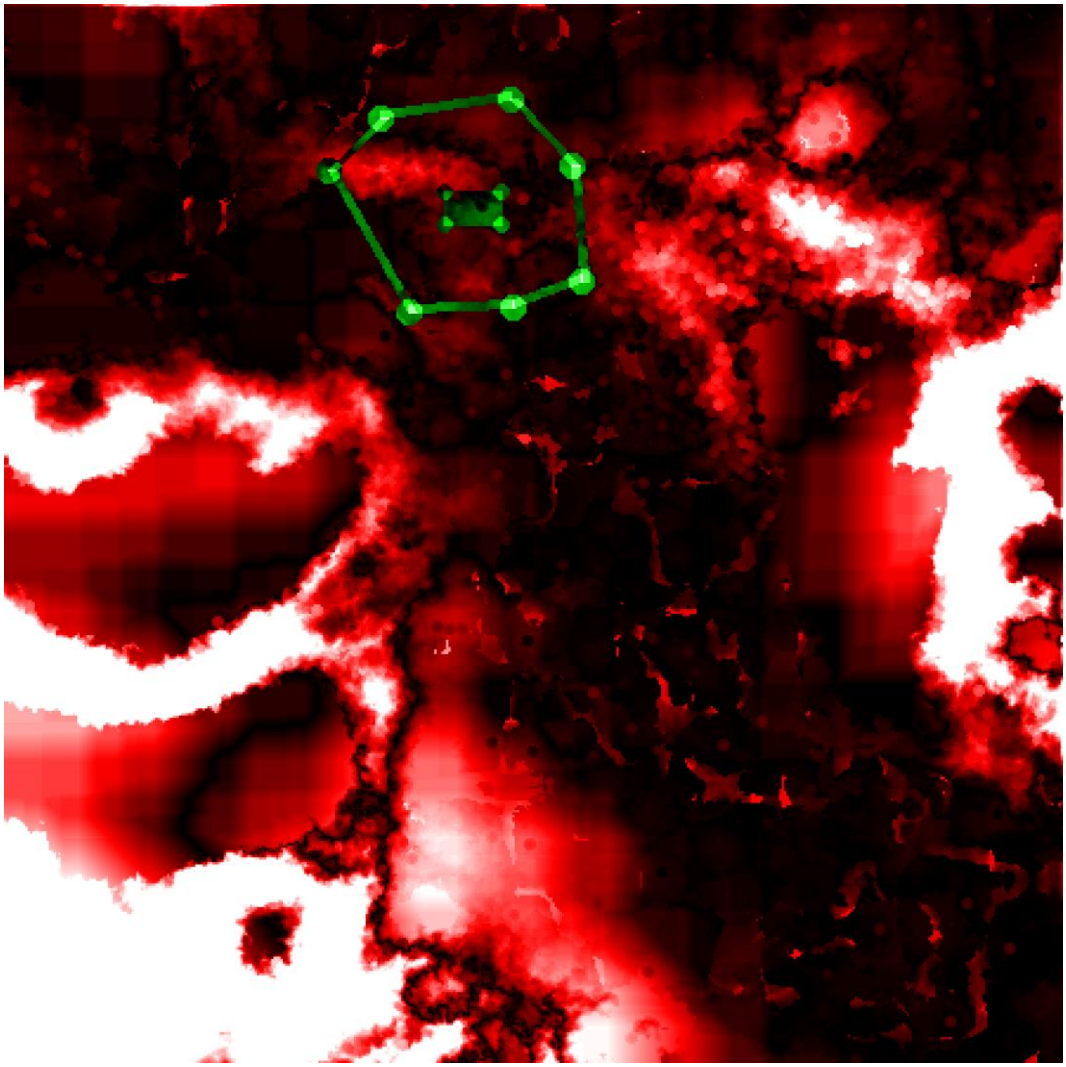


Figure 66 – Example 2 Terrain Fitness Map

Shown above is the terrain fitness map for this terrain. Note that it is rotated about 180° compared to the other screenshots. The problematic tower is the one furthest to the left in this image. It can be seen that it is moved to try to avoid the area of poor terrain fitness generated by the hill cutting roughly through the middle of the castle plan. The dip in the wall sections caused by this tower is problematic and would cause serious issues in the defensibility of the castle. Worst of all: it simply looks wrong. A further heuristic could be added to the castle layout algorithm to avoid placing towers significantly lower than its neighbours to mitigate this type of issue, or the algorithm may be modified so as to avoid water completely.

5.3 Example 3

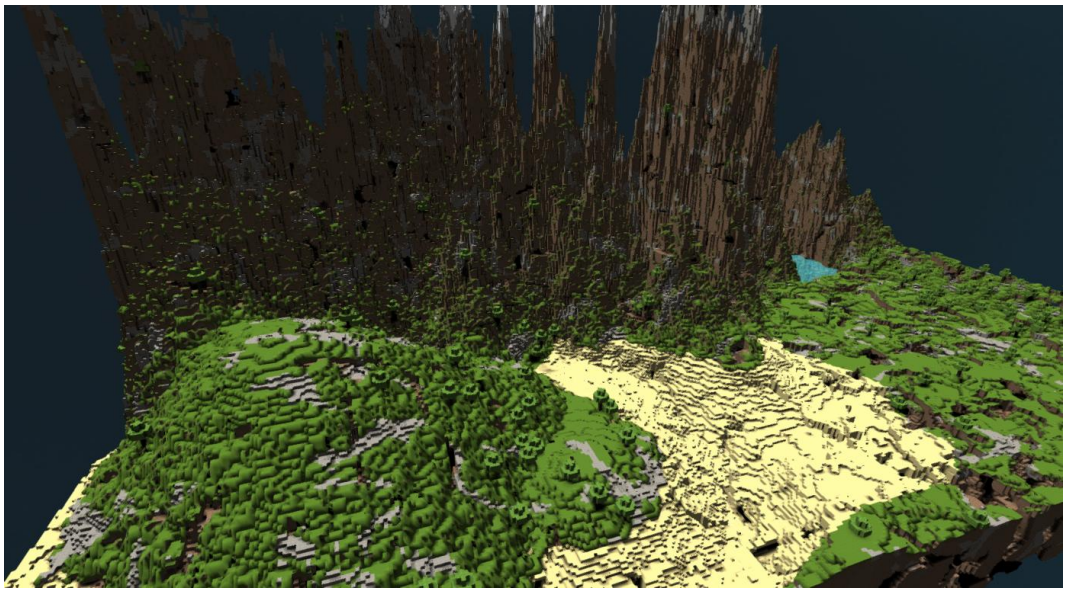


Figure 67 – Example 3 Terrain

This terrain (see Figure 67 above) has a large mountain range at the back, a big hill and some flat terrain with a desert. Ideally the castle should probably be built on the hill or into the mountain range.

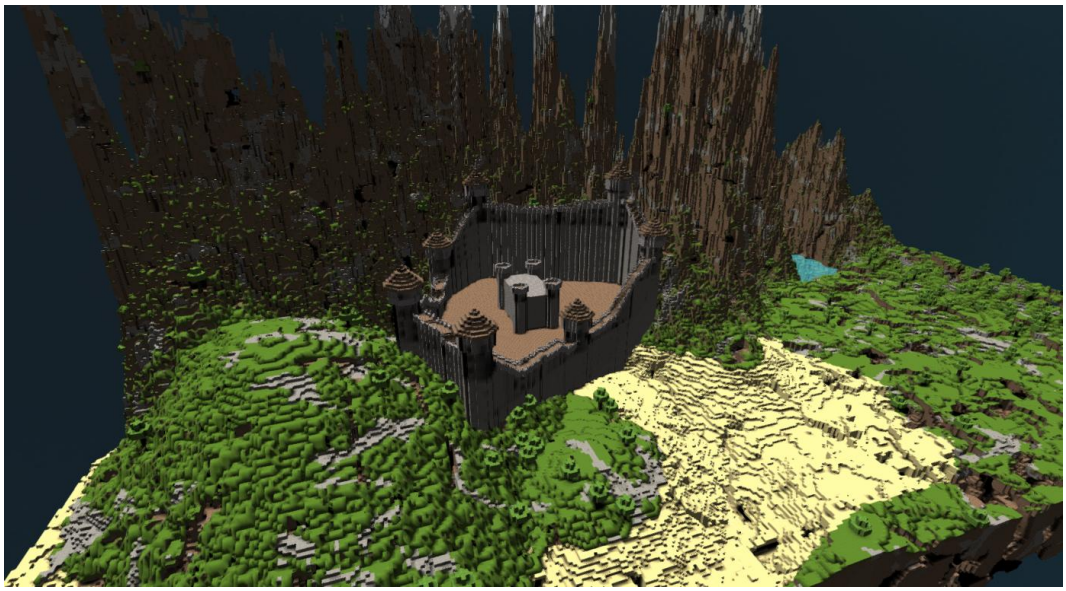


Figure 68 – Example 3 Castle

As can be seen in the image above (Figure 68) the castle is not placed on the hill but rather set into the mountain range at the back. It actually sits quite well into a concave area in the mountain, creating a strong defensive position to the back, while avoiding building the castle into the really problematic (steep) areas of the mountain.

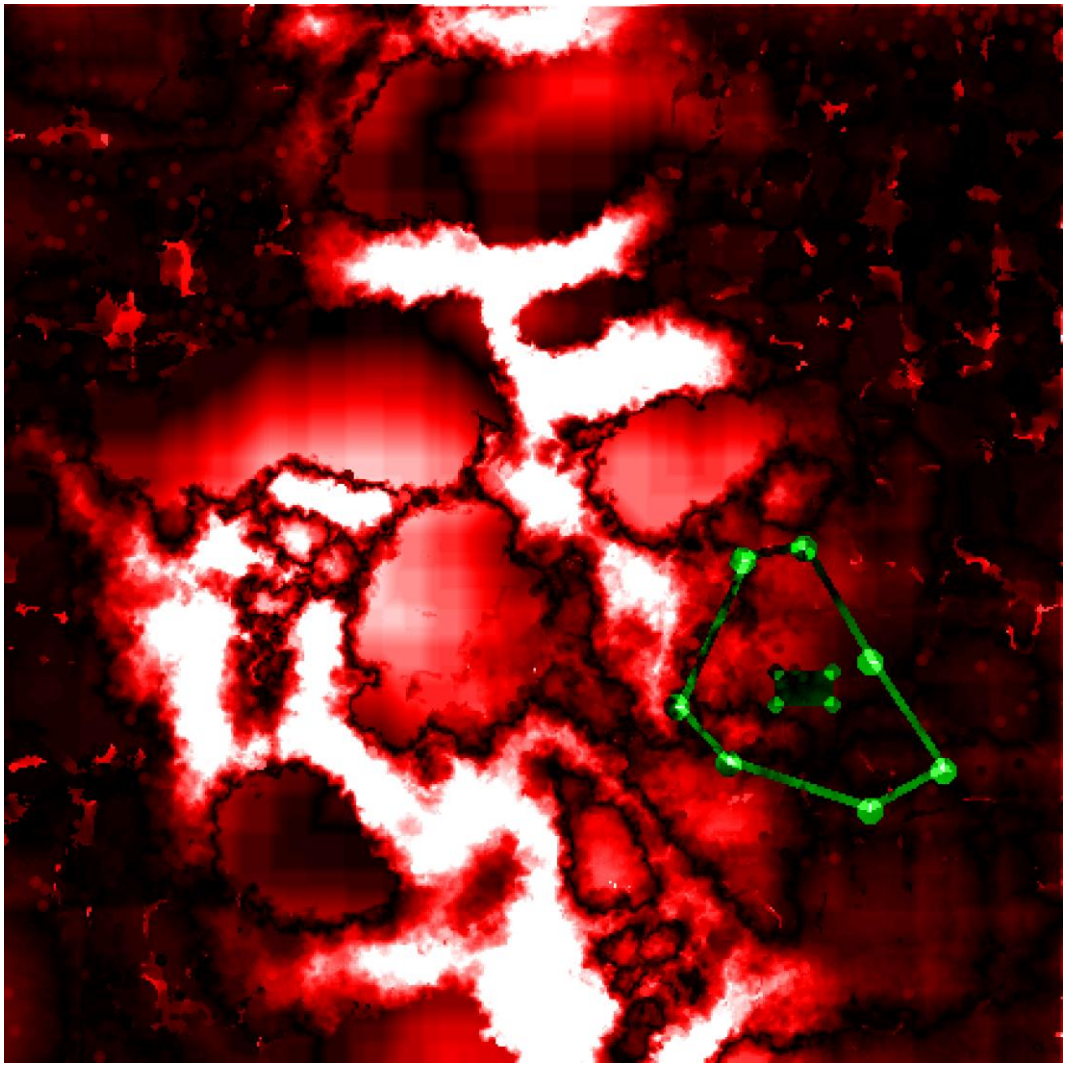


Figure 69 – Example 3 Terrain Fitness Map

The terrain fitness map above (Figure 69) shows that the castle nicely avoided the very problematic (strong white) areas of the mountain range, building into it as far as possible to gain height but not so far that it would cause major problems to the wall construction.

5.4 Example 4

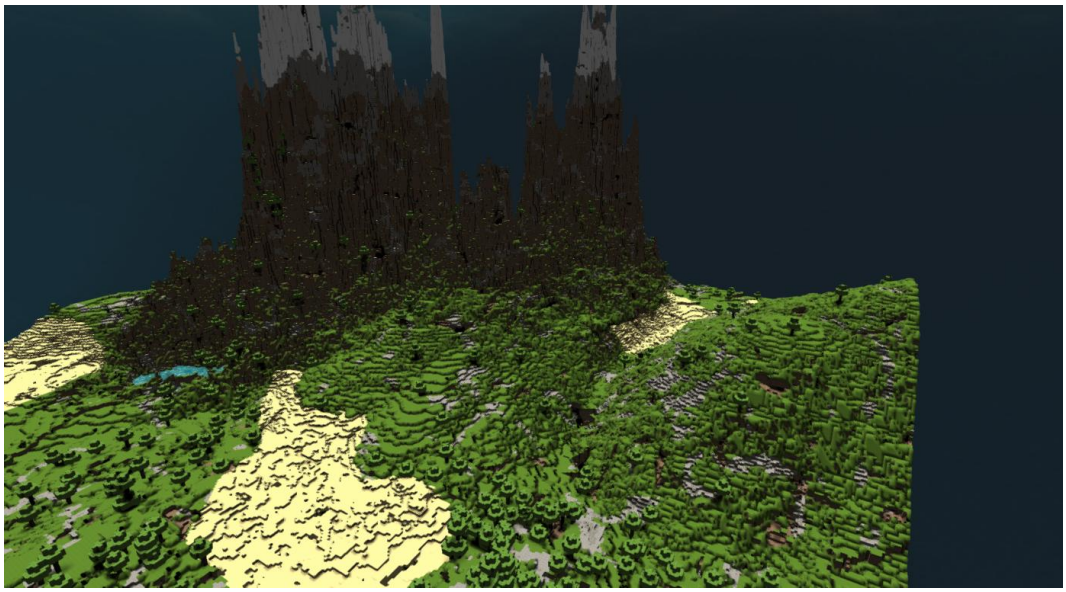


Figure 70 – Example 4 Terrain

This terrain (seen in Figure 70 above) features a large mountain and a large hill, separated by a valley. Expected placement might be on top of the hill or set into the front of the mountain.

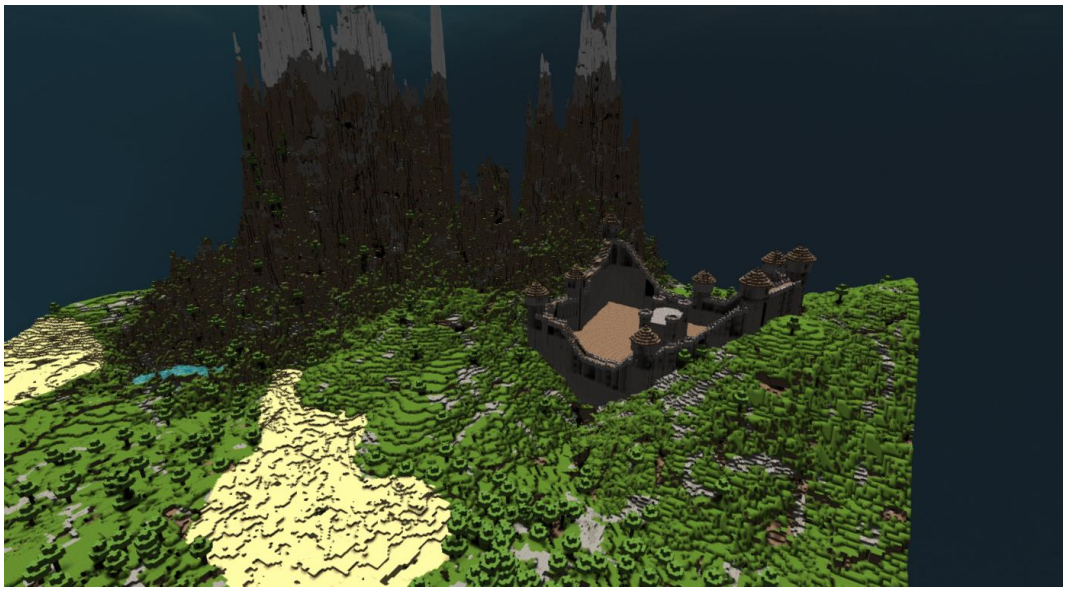


Figure 71 – Example 4 Castle

As can be seen in the image above (Figure 71) instead the algorithm placed the castle partially into the base of the mountain and partly into the hill, while spanning right across the valley. This actually makes a lot of sense both strategically and from how the algorithm works. Because the initial placement algorithm calculates the overall fitness only from the placement of the towers, the

valley area would be largely ignored as long as most of the towers are on the mountain base or the hill. Despite defying my instincts on where the castle should be placed, the final result can be considered acceptable.

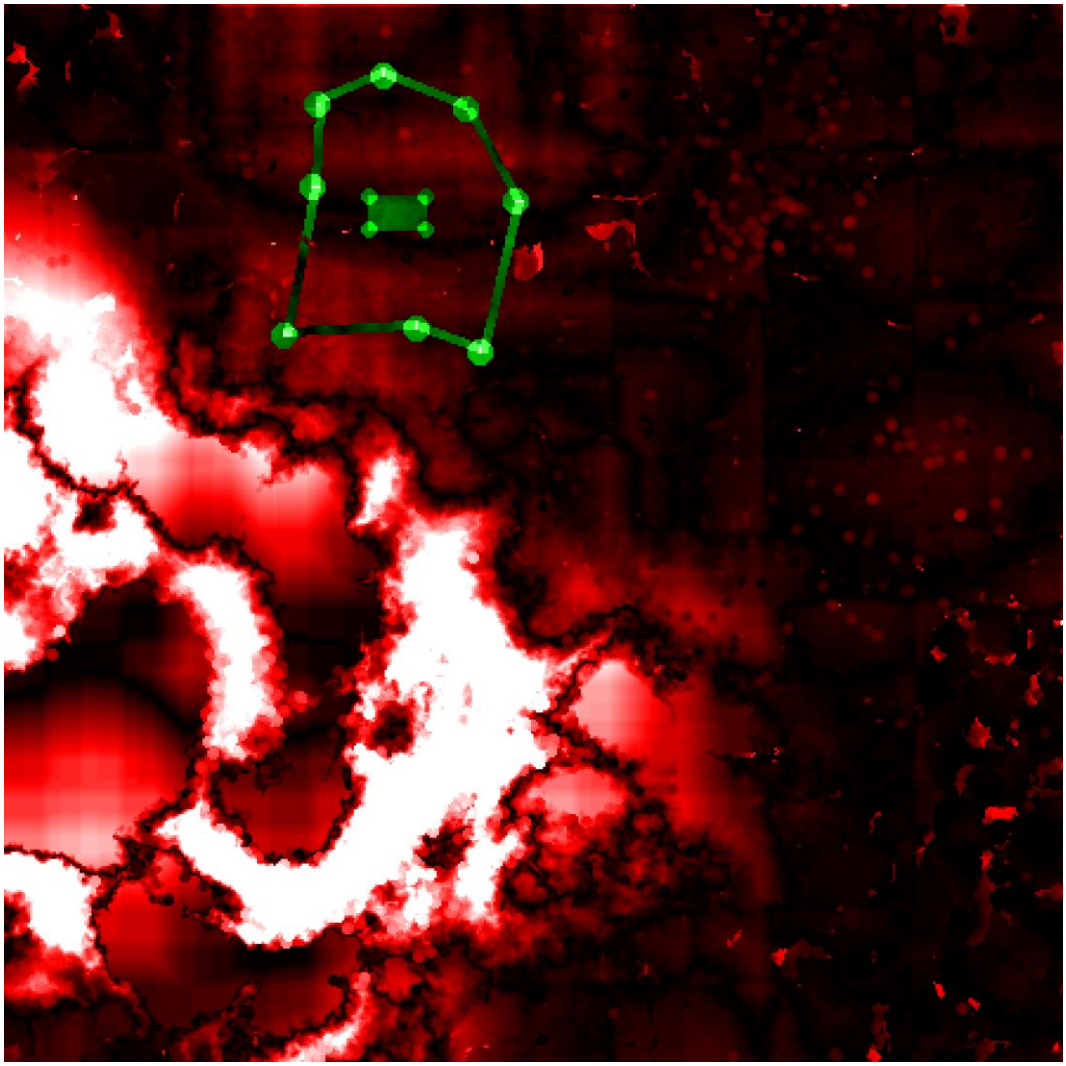


Figure 72 – Example 4 Terrain Fitness Map

As can be seen in the terrain fitness map above (Figure 72) the towers are distributed towards the top and bottom, with a larger gap between the two areas. This corresponds with the valley area on the terrain. As expected, the algorithm favoured high ground areas for tower placement where possible.

5.5 Example 5

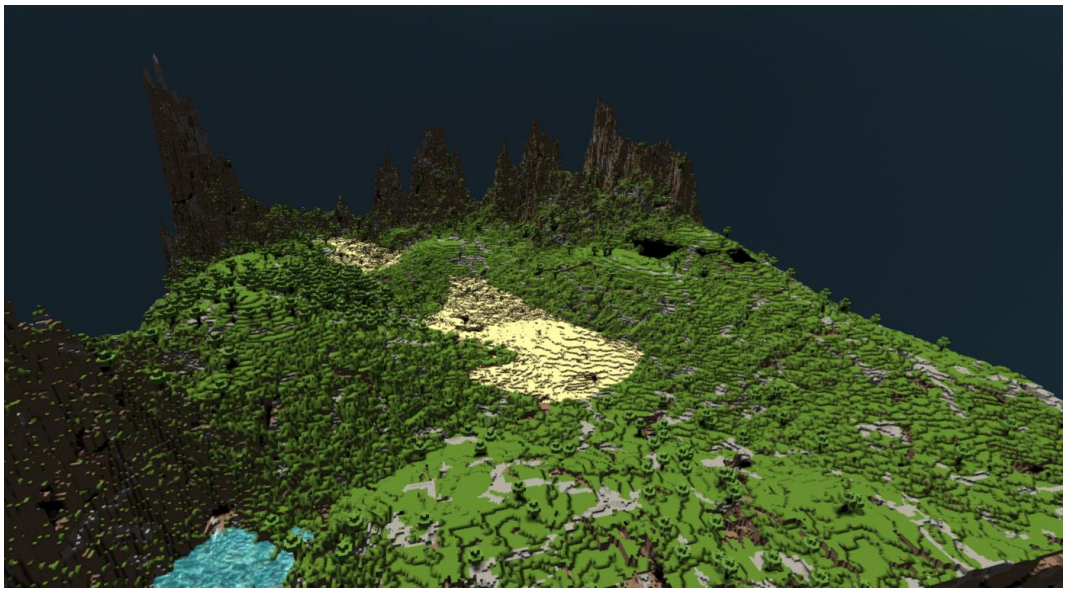


Figure 73 – Example 5 Terrain

The terrain presented above (Figure 73) shows complex rolling hills. There are a few possible sites for a castle to go, either on the hill at the back to the left, or to the right. It could also go on the large hill at the front, but this would place it lower than much of the surrounding terrain, causing a potential defensive weakness (i.e. it could be attacked with siege engines from the higher hills at the back, as discussed in the literature review section 2.1 Medieval Castle Construction).

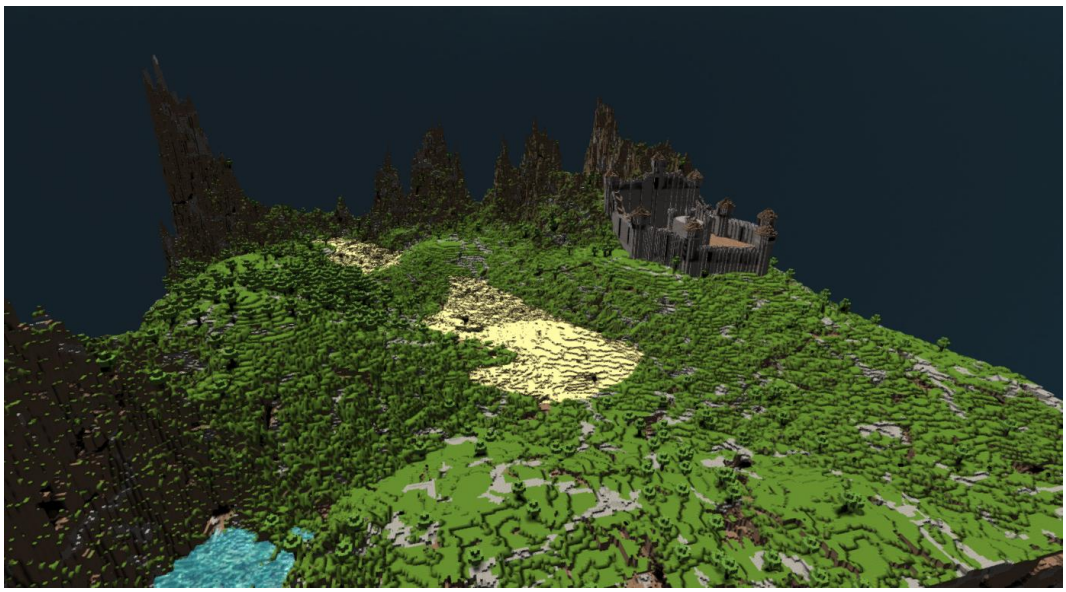


Figure 74 – Example 5 Castle

The castle generated for this terrain (see Figure 74) was placed high on one of the hilltops. On the back side of the castle it is set somewhat into the more mountainous area. Overall the result is more or less as expected and works well. A slight improvement would be if the back wall was not set quite so high into the mountainous area. As it is, the back wall has ended up somewhat elevated over the rest of the castle. This is not a major problem but a small amount of tweaking to the overall fitness score calculation might yield better results in cases like this.

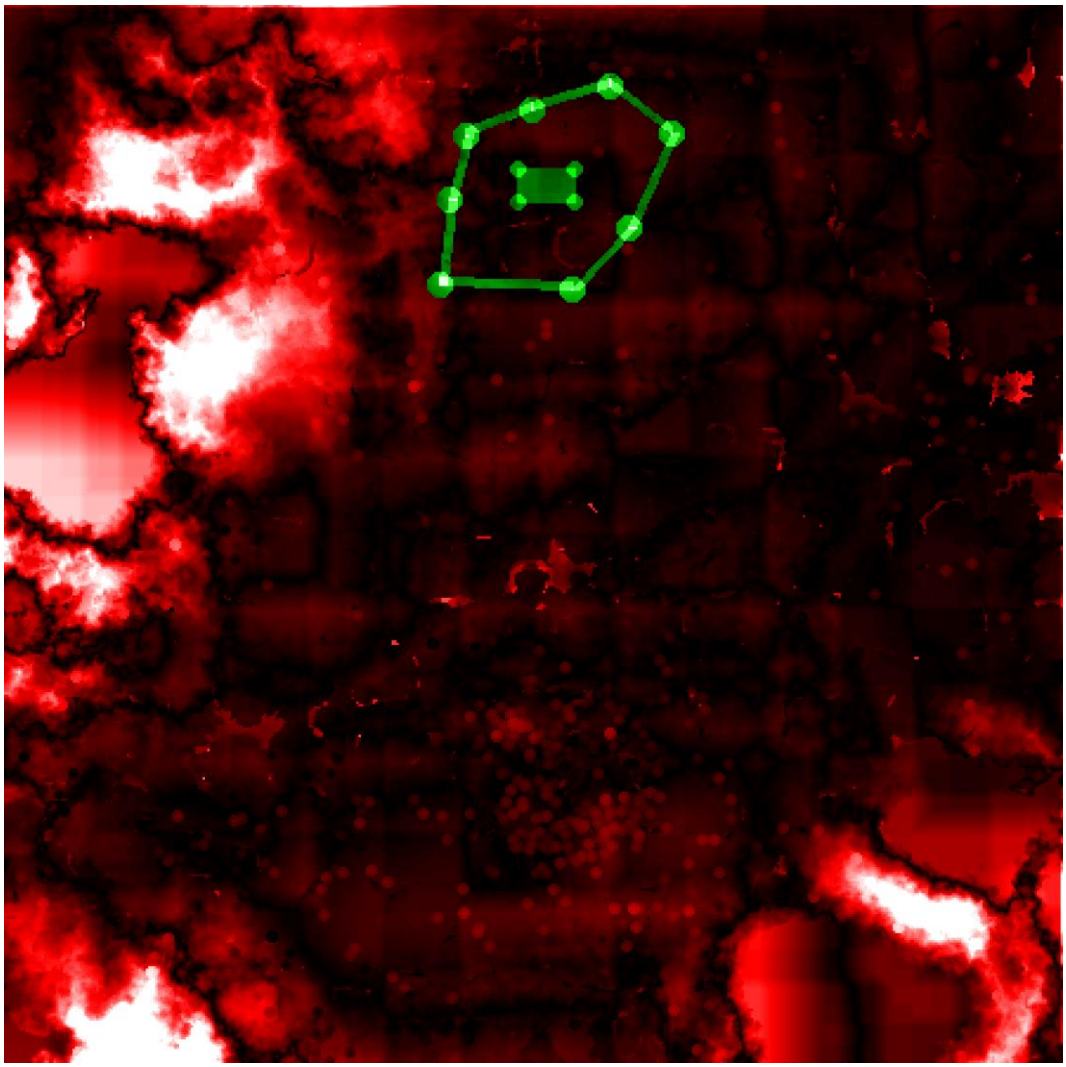


Figure 75 – Example 5 Terrain Fitness Map

When looking at the terrain fitness map (Figure 75) it can be seen that the algorithm has placed the castle as high as possible, while avoiding the problematic mountainous area on the left. Although the large area in the middle also has good terrain fitness values, i.e. the area is mostly dark on the map, the castle was correctly placed into a position occupying high ground.

Chapter 6: Evaluation

6.1 User Study

In addition to analyzing the generated castles myself I also conducted an online survey to test the ability of the algorithm to generate castles that look plausible to regular gamers familiar with Minecraft, and for it to do so for a large variety of terrains. For this purpose 100 terrains were generated with random seeds, and the algorithm was used to generate a castle on each (see Appendix C for a selection of castles generated for the survey). For every terrain and castle three screenshots were taken: one of the terrain and castle from a bird's eye view, the same view with just the terrain and no castle, and one screenshot from the ground (representing how the castle might look in an actual gameplay situation).

An online survey was then created with the images, using the Qualtrics online software package. The survey was set up so that every correspondent would see 5 different terrains with castles. This was deemed a good number[9] to keep the survey brief enough for the correspondents, while still returning enough data to be useful. For each castle the correspondents were asked to rate the overall appearance on a scale from 1 to 5 (the options were: "Very Bad", "Bad", "Neither Good nor Bad", "Good", and "Very Good"). They were also asked to rate how well they thought the castle was adapted to the terrain it was shown in, also on the same 5-point scale. Five options were considered enough that respondents could give nuanced answers (a choice between "Good" and "Very Good" for example) while not being overwhelming.

Once ethics approval was gained from the University of Waikato Ethics Panel (see Appendix D for the ethics approval letter) the survey was made available to members of the online Minecraft community through various channels (including Twitch.tv, social media and Internet forums). Respondents with expert knowledge

[9] Initially it was proposed that each correspondent would see 10 castle/terrain combinations in the survey. This was reduced down to 5 due to concerns by the author that some correspondents might abandon the survey early, or click through the final questions without proper consideration, if it was too long.

of Minecraft were chosen to avoid getting overly negative answers from people put off by the blocky nature of the geometry depicted (someone unaccustomed to the aesthetics of voxel world games might give low scores for the overall look of the castle simply because it is blocky, not realising this is a desired feature in this case).

The results of the study can be seen visualized as pie charts in Figure 76 and Figure 77 below.

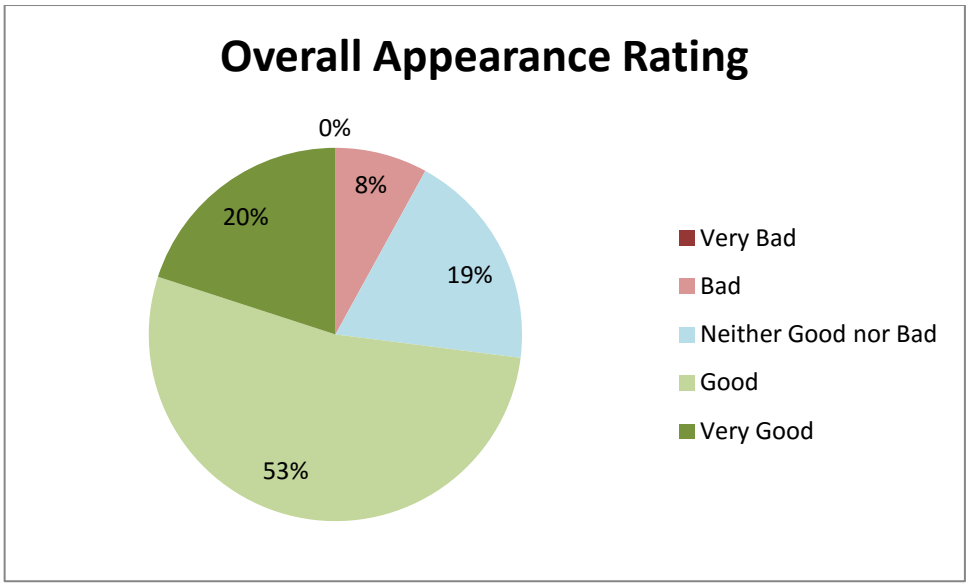


Figure 76 – User Study Result Overall Appearance

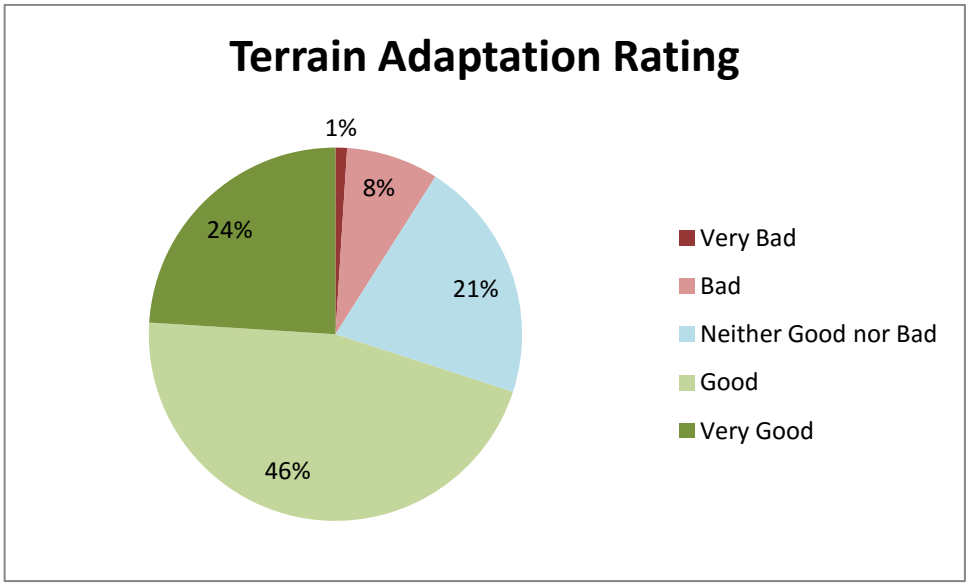


Figure 77 – User Study Result Terrain Adaptation

As can be seen in the results most of the responses were positive for both the overall appearance and the terrain adaptation. Less than 10% of responses in each

case were “Bad” or “Very Bad”, showing that the algorithm does a good job in the majority of cases. 73% of castles were judged as having “Good” or “Very Good” overall appearance, and 70% had “Good” or “Very Good” adaption to the terrain. For nearly all responses both questions were answered similarly (within 1 point of each other). In fact only in 3 out of the 100 total cases was there a bigger gap (in all 3 cases one question was answered with “Good” and the other “Bad” for a 2 point difference). This confirms what was already suspected: the overall appearance of the castle and how it is adapted to the terrain are strongly linked. Since the variation in the castles is driven by the terrain this is to be expected.

We will now look at a castle that was rated “Good” and one that was rated “Bad” and speculate as to why the study participants may have given those ratings. Figure 78 below shows a castle that received an overall “Good” rating across both questions. We can see that the castle has been created close to a high mountain at the back. Because the mountain is very steep it provides a defensive advantage at there and is unlikely to be able to be used by attackers as high ground for stone throwing siege weapons like trebuchets (the respondents may not have come to that conclusion via this reasoning but rather may have used an intuitive sense of this instead).

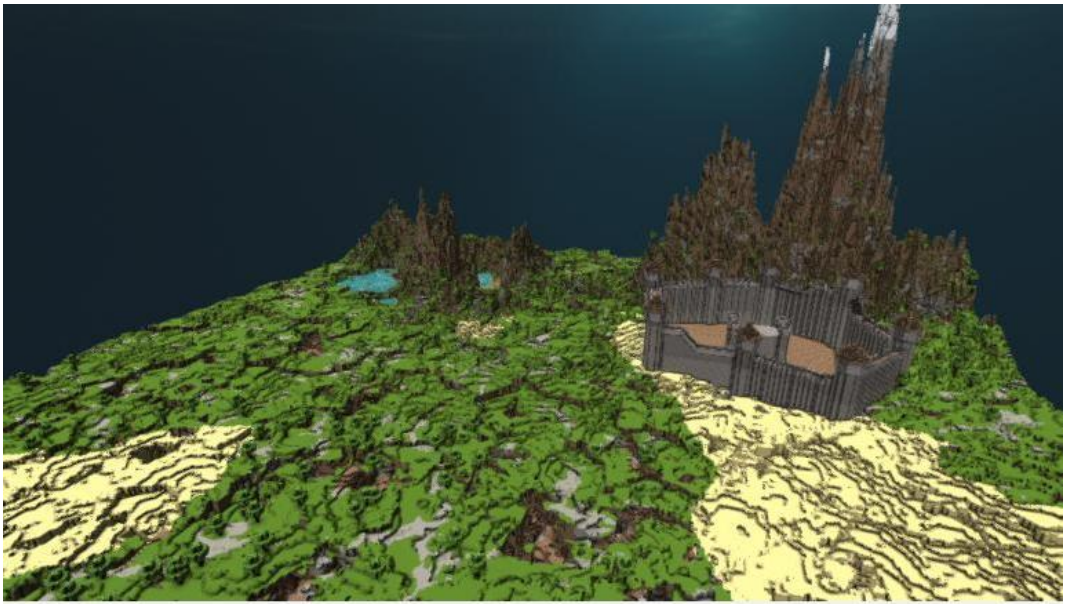


Figure 78 – Castle with a “Good” Rating for Both Questions

In Figure 79 shown below we can see an example of a castle that got a “Bad” rating for both overall appearance and terrain adaptation. We can see that this

castle was also built with a mountain at the back but with two crucial differences: firstly it is built right into the mountain with the back castle walls set into the mountain. Secondly the mountain here is rather small (it is just the outer reaches of the larger mountain range on the right) and not very steep. It is easy to see how attackers might be able to use this mountain to climb right onto the walls, obviating the need for scaling ladders or ropes (a technique of siege warfare discussed in section 2.1 Medieval Castle Construction). The hill on the left of the castle even overlooks some of the wall at the front, making it a perfect place to use siege machines to lob stones into the courtyard. In this case the algorithm appears to have favoured the high ground created by the mountain range outlier too highly. Adjusting the overall fitness score calculation (described in section 4.5.2.1 Initial Placement) slightly may be enough to resolve cases such as this.

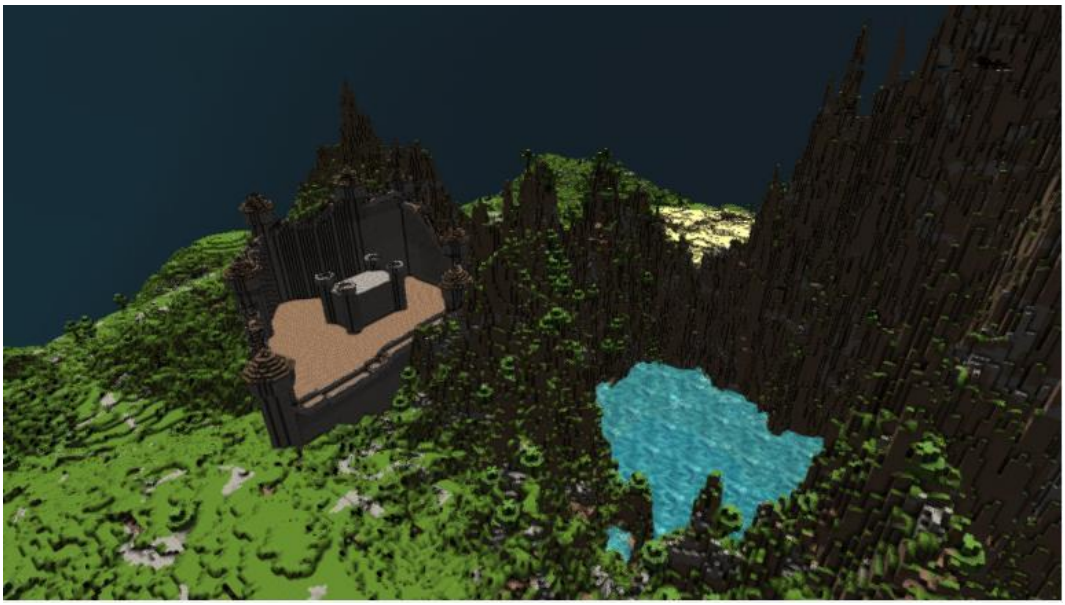


Figure 79 – Castle with a "Bad" Rating for Both Questions

While the result of the study is encouraging, it also shows that there are likely still some issues that would need to be resolved before the algorithm would be ready to be deployed in an actual game. Although we can speculate as to why respondents felt that some of the castles were badly generated, it might be useful to do a further study that could gain insight on what specific reasons respondents had for choosing the "Bad" option (i.e. what improvements they feel could be made to the algorithm). This input could then be used to modify the algorithm to provide a better result.

6.2 Evaluation of Results

Having looked at a broad range of output from this project (the examples shown in Chapter 5, all 100 castles generated for the user study, and others created during testing) I can make some judgements as to how successful the algorithm presented in this thesis is.

How the castles look is tied directly to the terrain so first I will comment on the terrain generator (described in section 4.2 Terrain Generation). The range of terrains generated is somewhat limited to different combinations of the three major basic terrain types implemented (mountains, rolling hills and flat grassland). After looking at many of them a certain amount of repetition in the overall look is inevitable. This was done by design: in order to be able to test the castle placement algorithm I would need to be able to look at a representative sample of terrains to ensure that the algorithm performed well. A more complex terrain generator would have created a much broader variation in terrain, meaning far more testing would be needed (this is an important caveat to using the castle generation algorithm in a real voxel world game: it would need significant testing and tuning to ensure good results with a different terrain generator). So for the purposes of this project I think that the terrain generator produced good results. There is enough variation to provide a good challenge for castle placement and to create and number of different and interesting castle configurations.

The results of the castle placement and layout generation are mixed. In most cases the castle will be well adapted to the terrain. Very problematic areas (for example the sides of steep mountains) are avoided and the castle usually will occupy a high ground position if possible. Even though the castle generation process is entirely deterministic and always starts from the same basic layout, the adaptation to the terrain is sufficient to create a number of different castles. However, there remain some terrains where the castle generation algorithm produces sub-optimal results (based on the user study this would be around 8% of the time; see section 6.1 User Study). Overall I believe the castle generation algorithm is sound, it just needs a little tuning to deal with those edge cases that still present problems.

Chapter 7: Conclusion

7.1 Review

This research aimed to explore the possibility of procedurally generating large and complex structures like castles in voxel world games and how they could be adapted to randomly generated terrain.

As discussed in the literature review, consideration was taken for how medieval castles were constructed and what defensive features would need to be addressed in the context of siege warfare to create a good castle. Existing techniques were for building procedural architecture were considered and the most useful (search-and-optimize) was adapted for our purpose. Ways of smoothing surfaces created from voxel data were examined and marching cubes was used as inspiration for a simple algorithm to smooth parts of the terrain. Existing voxel world games were studied to see what had been done, for inspiration on how to implement a voxel world, and to see what is possible to create with voxels.

Four major components to the project were identified in the project design chapter: a voxel world engine, terrain generation, castle placement, and castle construction.

How these components were created was explained in the implementation chapter. A voxel world engine was created on top of Unity3D (a general purpose game engine) to create a sandbox that could be used to build the other parts of the project in. The engine made use of chunks to store the voxels and divide them up, allowing the voxel world bounds to grow dynamically (for example to add more height for higher mountains). It implemented a simple lighting algorithm capable of creating fast approximate global illumination. For display, the engine would convert the voxels to polygons, being capable of producing basic cubes but also more elaborate shapes. It also introduced a feature novel to voxel world engines: layers. By adapting a layer system like that found in 2D drawing packages to a

voxel system it allowed the ability to hide/show voxel elements at will, making it easy to compare the terrain with and without a castle on it.

Terrain generation was implemented using a variety of noise functions linked together to create a plethora of terrains that castles could be built upon. It first creates a heightmap that is used in conjunction with additional noise functions to calculate the terrain blocks placed into the world. Additional steps in the terrain generation process add trees and water, as well as smoothing parts of the terrain using a simplified adaptation of the marching cubes algorithm.

In order to efficiently place blocks into the world to create buildings 2D drawing functions were adapted to work in a voxel engine. These were combined with a brush system to allow the flexible manipulation of the voxel world.

With all the basic tools in place: a voxel world engine, randomly generated terrain, and system for working with voxels easily (the drawing system with the brushes), it was now possible to create an algorithm to generate castles. First the terrain was analyzed and the terrain fitness calculated. The map of terrain fitness values was then used to search for the best place to create the castle and it was used to optimize the layout of the castle to fit into the terrain.

With the location and layout calculated all that remained was the final step: actually constructing the castle in the voxel world. The walls and towers were generated at the perimeter and the inside area was cleared to create a courtyard. Then a keep was generated in the centre of the courtyard area.

The demonstration chapter of this thesis showed five different terrains and the castles that were generated in them, as well as the terrain fitness maps that they used during the placement and layout optimization process. It discussed where the algorithm worked well and where it could have done better.

A user study was carried out via online survey to get a broader picture of how well the algorithm performed at creating castles suitable to be used in a voxel world game. This showed that most castles were acceptable but that in just under 10% of cases the result of the castle generation algorithm was not good enough. In addition to the user study the results of the algorithm were evaluated by the author. It was concluded that some edge cases do indeed produce poor results

from the castle generation algorithm but that this could be fixed simply by tuning the algorithm a little more for those cases, and that fundamentally the algorithm is sound.

7.2 Future Work

There are a number of areas that could be expanded in future work on this subject. There are still some terrains where the algorithm produces suboptimal results (for example see 5.2 Example 2 and Figure 79 in the section 6.1 User Study). Some further small tweaks to the placement and layout algorithm should be able to improve results for most of the remaining problem terrains.

A large amount of work could be done to vary the generated castles. For example more tower types could be added and a set could be chosen randomly when the castle is created. This would help greatly in creating a more varied look for the castles. Even better would be if the castle-look varied depending on the surrounding terrain, for example using sandstone in the desert, or a more wood-heavy construction when near a forest.

The keep could use a lot of work to make it look more interesting. Perhaps a project in itself would be to procedurally generate the keep, including the interior. Additionally, some castles have no keep but rather use a large tower, often set into the outer curtain wall, so the castle constructor could also randomly create this. Furthermore, the entire courtyard area could be procedurally filled with a variety of auxiliary buildings such as stables, an armoury or a chapel. Another consideration could be to create a more complicated castle layout with a second wall interior to the outer curtain wall for a defence in depth approach that some castles used.

This work could also be expanded to generate a whole medieval fortified town. The outer wall could be generated in the same fashion as the wall is created for the castles in this project, only on a larger scale. The inside area could then be populated with procedurally generated buildings, on a procedurally generated town plan. A step further could even create a whole populated area with a town, a castle, farms, and a road network connecting them all together.

Bibliography

Aleksandr. (2014, September 3). *Documentation, Unity scripting and you - Unity Blog*. Retrieved June 10, 2015, from Unity3d.com: http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/

Bear, S. C. (2013). *Ten Epic Minecraft Castles For Inspiration*. Retrieved July 9, 2015, from Mincraft Pixel Art Ideas: http://minecraftpixelartbuildingideas.blogspot.co.nz/2013/02/ten-epic-minecraft-castles.html

Bevins, J. (2005a). *noise::module::Billow Class Reference*. Retrieved June 24, 2015, from Libnoise: http://libnoise.sourceforge.net/docs/classnoise_1_1module_1_1Billow.html

Bevins, J. (2005b). *noise::module::RidgedMulti Class Reference*. Retrieved June 24, 2015, from Libnoise: http://libnoise.sourceforge.net/docs/classnoise_1_1module_1_1RidgedMulti.html

Bevins, J. (2005c). *What is coherent noise?* Retrieved June 24, 2015, from Libnoise: http://libnoise.sourceforge.net/coherentnoise/index.html

Cepero, M. (2015, 06 26). *Procedural World: Castle by the lake*. Retrieved July 14, 2015, from Procedural World: http://procworld.blogspot.co.nz/2015/06/castle-by-lake.html

Cepero, M. (2013, August 3). *Procedural World: EverQuest Next*. Retrieved June 30, 2015, from Procedural World: http://procworld.blogspot.co.nz/2013/08/everquest-next.html

Cepero, M. (2010, November 19). *Procedural World: From Voxels to Polygons*. Retrieved June 30, 2015, from Procedural World: http://procworld.blogspot.co.nz/2010/11/from-voxels-to-polygons.html

Comanche: Maximum Overkill screenshots for DOS - MobyGames. (n.d.). Retrieved July 2, 2015, from Moby Games:

http://www.mobygames.com/game/dos/comanche-maximum-overkill/screenshots/gameShotId,242958/

Crassin, C., Neyret, F., Lefebvre, S., & Eisemann, E. (2009). GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. *I3D 2009 conference proceedings.*

Crassin, C., Neyret, F., Lefebvre, S., Eisemann, E., & Sainz, M. (2009). *Beyond Triangles Giga Voxels Effects in Video Games*. Retrieved June 30, 2015, from Models and Algorithms for Visualization and Rendering: http://maverick.inria.fr/Publications/2009/CNLSE09/GigaVoxels_Siggraph09_Slides.pdf

Crytek. (2014, January 16). *Voxel Objects*. Retrieved June 30, 2015, from CryEngine Manual: http://docs.cryengine.com/display/SDKDOC2/Voxel+Objects

Daybreak Game Company LLC. (2015). *Landmark - Landmark FAQ*. Retrieved July 1, 2015, from Landmark: https://www.landmarkthegame.com/landmark-faq

EQNexus. (2013, October 11). *Voxels and EverQuest Next: An Interview with Steve Klug*. Retrieved June 30, 2015, from EQNexus: http://eqnexus.com/2013/10/voxels-everquest-next-interview-steve-klug/

Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (2001). *Computer Graphics Principles and Practice.* Addison-Wesley Publishing Company.

Form 8-K: Bankruptcy or receivership. (2002, October 21). Retrieved June 30, 2015, from SEC Archive: http://www.sec.gov/Archives/edgar/data/1010026/000095013402012752/d00519e8vk.txt

Gravett, C., & Hook, A. (2007). *The Castles Castles of Edward I in Wales 1277-1307.* New York: Osprey Publishing.

Haas, P. (2013, August 3). *EverQuest Next Could Come To PS4*. Retrieved June 30, 2015, from Cinemablend: http://www.cinemablend.com/games/EverQuest-Next-Could-Come-PS4-58039.html

Hohmann, B., Krispel, U., Havemann, S., & Fellner, D. (2009). CityFit: High-quality urban reconstructions by fitting shape grammars to images and derived textured point clouds. *Proceedings of the 3rd ISPRS Workshop.*

Ju, T., Losasso, F., Schaefer, S., & Warren, J. (2002). Dual Contouring of Hermite Data. *SIGGRAPH '02 Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, (pp. 339-346). New York, NY, USA.

Khaw, C. (2013, August 8). *Indies Did This: How Voxel Farm and StoryBricks are Helping to Shape Everquest Next*. Retrieved June 30, 2015, from USgamer: http://www.usgamer.net/articles/-indies-did-this-how-voxel-farm-and-storybricks-are-helping-to-shape-everquest-next-

Laine, S., & Karras, T. (2010). *Efficient Sparse Voxel Octrees - Analysis, Extensions, and Implementation.* NVIDIA.

Light - Minecraft Wiki. (n.d.). Retrieved July 1, 2015, from Minecraft Wiki: http://minecraft.gamepedia.com/Light

Lorensen, W. E., & Cline, H. E. (1987). Marching Cubes: A High Resolution Surface Construction Algorithm. *ACM Computer Graphics* , 163-169.

Merrell, P., Schkufza, E., & Koltun, V. (2010). Computer-generated residential building layouts. *ACM Transactions on Graphics (TOG) - Proceedings of ACM* .

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Grool, L. (2006). Procedural Modeling of Buildings. *SIGGRAPH '06 ACM SIGGRAPH 2006 Papers* (pp. 614-623). New York: ACM.

Musgrave, K. F. (2002). Procedural Fractal Terrains. In D. S. Ebert, K. Musgrave, D. Peachey, K. Perlin, & S. Worley, *Texturing and Modeling a Procedural Approach* (pp. 489-506). Morgan Kaufmann.

Nielson, G. M., & Hamann, B. (1991). The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. *VIS '91 Proceedings of the 2nd conference on Visualization '91* (pp. 83-91). Los Alamitos, CA, USA: IEEE Computer Society Press.

NovaLogic Awarded Patent for Unique 3-D Graphics Engine. (2000, February 18). Retrieved June 30, 2015, from Business Wire: http://www.thefreelibrary.com/NovaLogic+Awarded+Patent+for+Unique+3-D+Graphics+Engine%3B+Voxel+Space...-a059539908

Nystrom, R. (2014). *Flyweight - Design Patterns Revisited - Game Programming Patterns*. Retrieved June 10, 2015, from Game Programming Patterns: http://gameprogrammingpatterns.com/flyweight.html

Parish, Y. I., & Müller, P. (2001). Procedural Modeling of Cities. *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 301-308). New York: ACM.

Peckham, M. (2014, September 15). *Minecraft Is Now Part of Microsoft, and It Only Cost $2.5 Billion*. Retrieved June 30, 2015, from Time: http://time.com/3377886/microsoft-buys-mojang/

Perlin, K. (1984). Advanced Image Synthesis. *ACM SIGGRAPH.*

Perlin, K. (1985). An Image Synthesizer. *Proceedings of ACM SIGGRAPH 85.*

Perlin, K., & Hoffert, E. M. (1989). Hypertexture. *Proceedings of ACM SIGGRAPH 89.*

Persson, M. (2011, March 10). *The Word of Notch — Terrain generation, Part 1*. Retrieved June 30, 2015, from Tumblr: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants.* New York: Springer-Verlag.

Silva, P. B., Müller, P., Bidarra, R., & Coelho, A. (2013). Node-based shape grammar representation and editing. *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games, co-located with the Eigth International Conference on the Foundations of Digital Games.*

Stokstad, M. (2005). *Medieval Castles.* Westport: Greenwood Press.

Stronghold - Minecraft Wiki. (n.d.). Retrieved July 9, 2015, from Minecraft Wiki: http://minecraft.gamepedia.com/Stronghold

Village - Minecraft Wiki. (n.d.). Retrieved June 4, 2015, from Minecraft Wiki: http://minecraft.gamepedia.com/Villages

Voxel - Landmark Wikia. (2014). Retrieved July 9, 2015, from Landmark Wikia: http://landmark.wikia.com/wiki/Voxel

Voxel Farm. (2015). *Voxel Farm*. Retrieved June 30, 2015, from Voxel Farm: http://voxelfarm.com/

Worley, S. (2002). Cellular Texturing. In D. S. Ebert, K. Musgrave, D. Peachey, K. Perlin, & S. Worley, *Texturing and Modeling a Procedural Approach* (pp. 135-155). Morgan Kaufmann.

Appendix A

This is a list of all the brushes with brief descriptions of what they do. Most of the important ones also have more detailed descriptions in the relevant sections that explain their operation. Note that a number of these functions reference the smooth heightmap – this is explained in more detail in section 4.4.1 The Smooth Heightmap. The brushes are broken up into two sections: those that output voxels to the world, and utility brushes that perform other functions.

Voxel Brushes

- `BulldozerBrush` – Removes blocks from the y position passed in down to the height of the smooth heightmap at this location. The block at the height of the smooth heightmap is set to a block type passed into the constructor for the brush.
- `CheckerBrush` – Outputs a checker pattern of voxels. This is similar to the `StippleBrush` in that it essentially outputs a voxel for every second voxel (and does nothing for the other). However, it will always maintain a checker pattern by using the following pseudocode:

```
if( x % 2 == 0 && z % 2 != 0 )
  OutputVoxel();
else if( x % 2 != 0 && z % 2 == 0 )
  OutputVoxel();
```

- `FillBelowBrush` – Fills the area below the given y position value with a block type until a non-empty voxel is encountered or a certain limit is reached.
- `FillBelowBrushOnHeightmap` – Same as `FillBelowBrush` but uses the y position value as an offset from the smooth heightmap instead of the direct location.
- `FlatBulldozerBrush` – Removes blocks from the y position down to a given height value and places a single block of a given type at the lowest position.

- `MoatExcavatorBrush` – Used to create moats. Similar to the `FlatBulldozerBrush` but also keeps track of if it removed any water voxels. For more information on moats see section 4.5.3.5 Moats.
- `RandomSolidBrush` – Takes two block types and a probability value. Randomly chooses one block type or the other to output based on the probability value. Used to randomly add mossy stone blocks to the towers.
- `SolidBrush` – The simplest brush. Takes a single bock type in the constructor. Just outputs a single voxel at the position with the block type it is set to.
- `SolidBrushOnHeightmap` – Same as `SolidBrush` but uses the y position value as an offset from the smooth heightmap instead of the direct location.
- `StippleBrush` – Outputs a voxel every other time that `WriteVoxel()` is called.
- `StippleBrushOnHeightmap` – Same as `StippleBrush` except it uses the y position value as an offset from the smooth heightmap instead of the direct location.
- `WallBrush` – Creates a wall section at the given location. Used to create the walls. For more information see section 4.5.3.2 The Wall.

Utility Brushes

- `AreaDefenceCalculatorBrush` – Used to calculate the overall fitness score over an area. See section 4.5.2.1 Initial Placement for more information on how the overall fitness score is calculated.
- `DefenceAdderBrush` – Takes an integer value in the constructor and adds it to the terrain fitness map (see 4.4.2 Calculating the Terrain Fitness Score).
- `HeightmapperBrush` – Used to create a black and white 2D heightmap image.
- `InitialTowerPlacementOptimizerBrush` – Used during the initial tower placement (for more on initial tower placement and the overall fitness score see 4.5.2.1 Initial Placement). Calculates the overall fitness of the castle by summing together the fitness scores at each tower

position. It then checks if the overall fitness score calculated is better than the others calculated with this brush so far, and if it is the current castle location is stored as the best seen so far, and the score is also kept.

- `MinimapperBrush` – Used to create a minimap of the terrain. This takes the top voxel in the world at the given (x, z) location and writes it to an image with a simple colour mapping (e.g. water is blue, grass green, dirt brown and so on). This was used in early development to help visualize the world. An example output can be seen below this list (Figure 80).
- `PolygonBuilderBrush` – Used by the `FillConvexPolygon()` function to create a list of edge voxels. See section 4.3.1.3 Fill Polygon for more information on `FillConvexPolygon()`.
- `PreOptTowerPlacementBrush` – Used in the first step of tower placement optimization. For more information see section 4.5.2.2 Pre-Optimization Step.
- `SmoothHeightmapMinMaxFinderBrush` – Used to find the highest and lowest points of the smooth heightmap over an area. Also computes the average height over the area. Simply reads the smooth heightmap value at the given (x, z) location and compares it to the highest and lowest value encountered so far. If it is higher than the previous highest then this value becomes the new highest value, if it is lower than the previous lowest it becomes the new lowest. The value is also added to a running total and a count variable is incremented. When the average is requested $\frac{total}{count}$ is returned.
- `TerrainDefenceEvaluatorBrush` – Calculates the terrain fitness score at the given location and records it in the terrain fitness map. For more information on the terrain fitness score see section 4.4.2 Calculating the Terrain Fitness Score.
- `TowerPlacementOptimizerBrush` – Used during the tower placement optimization process. Draws a line to a given anchor point and calculates the terrain fitness along the line. For more information on this see the section on tower optimization (section 4.5.2.3 Optimization Step).

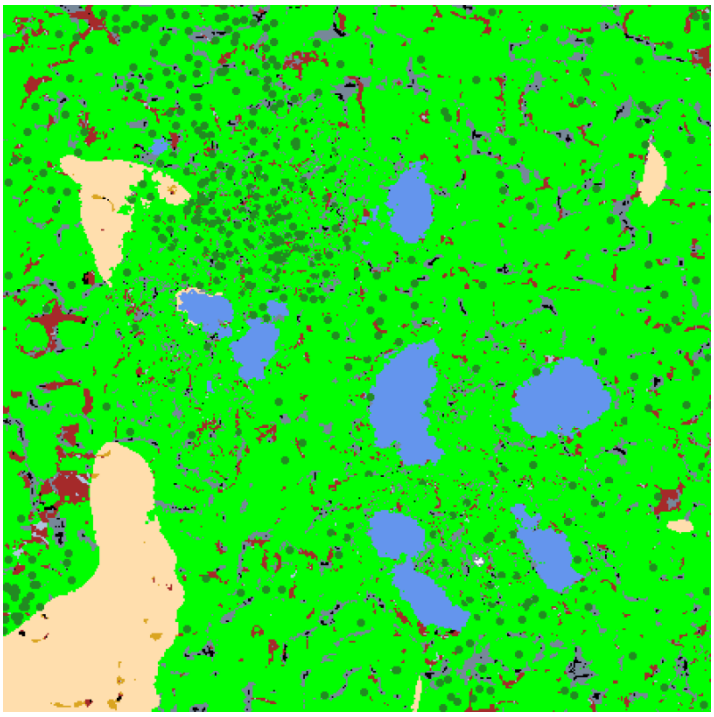


Figure 80 – A Minimap Created by the MinimapperBrush

Appendix B

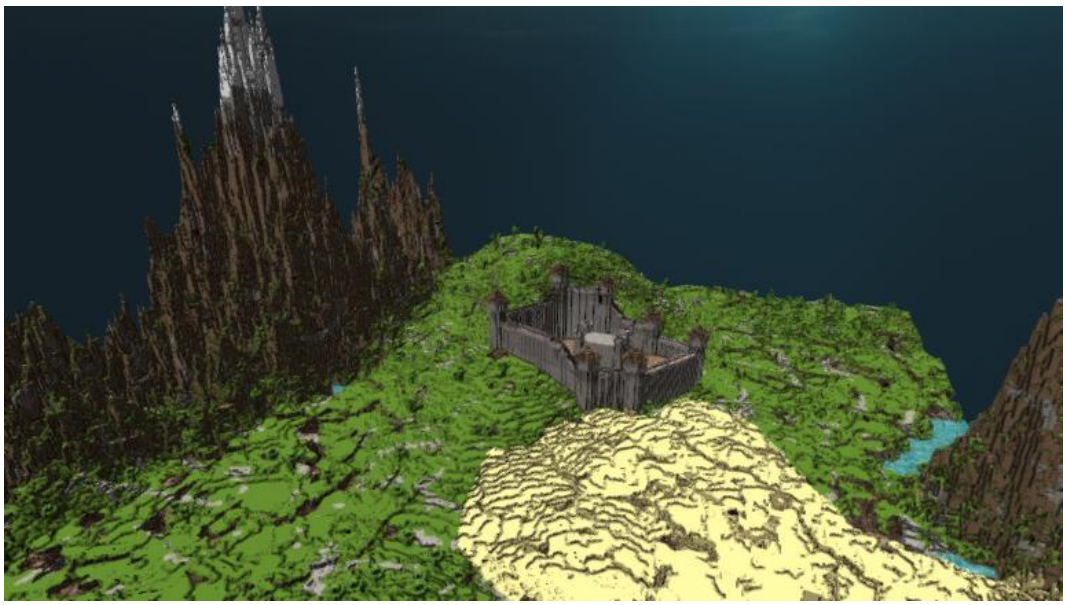
Values used during castle construction in the screenshots in this document.

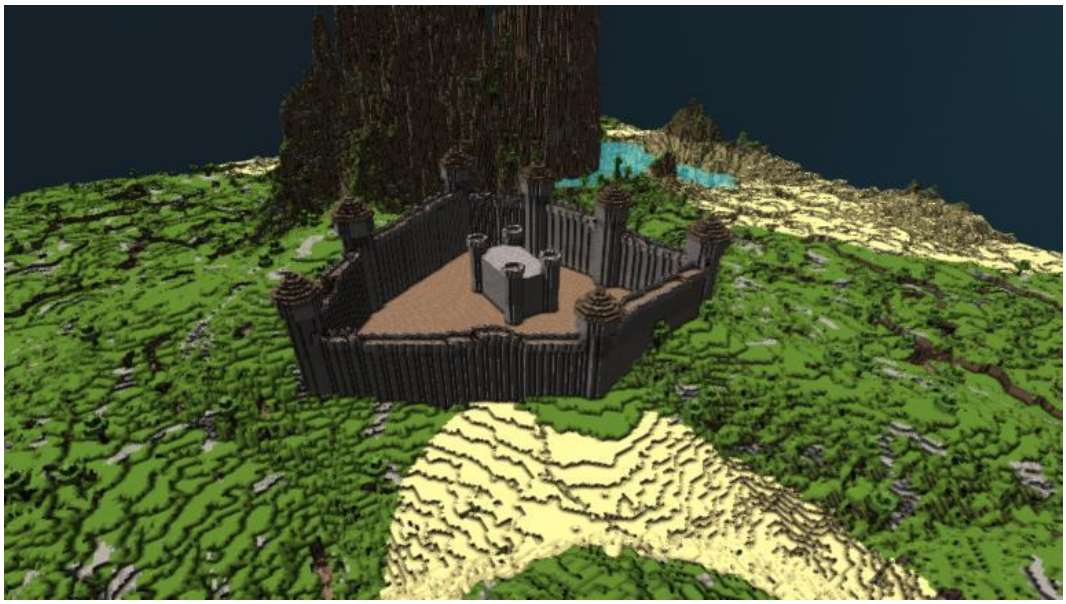
Position	-50, -50
Width	100
NumberOfTowers	10
OptIterations	1
InitialPlaceSearchRadius	140
PreOptSearchRadius	16
OptSearchRadius	8
BidirectionalSearch	False
TowerMergeRadius	24

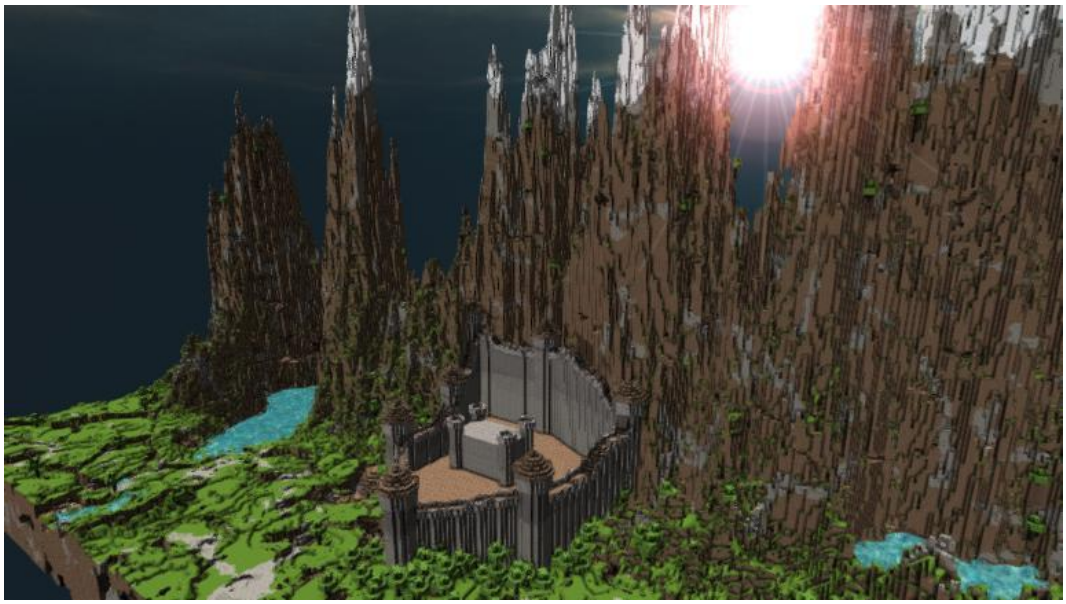
Table 2 - Values Used

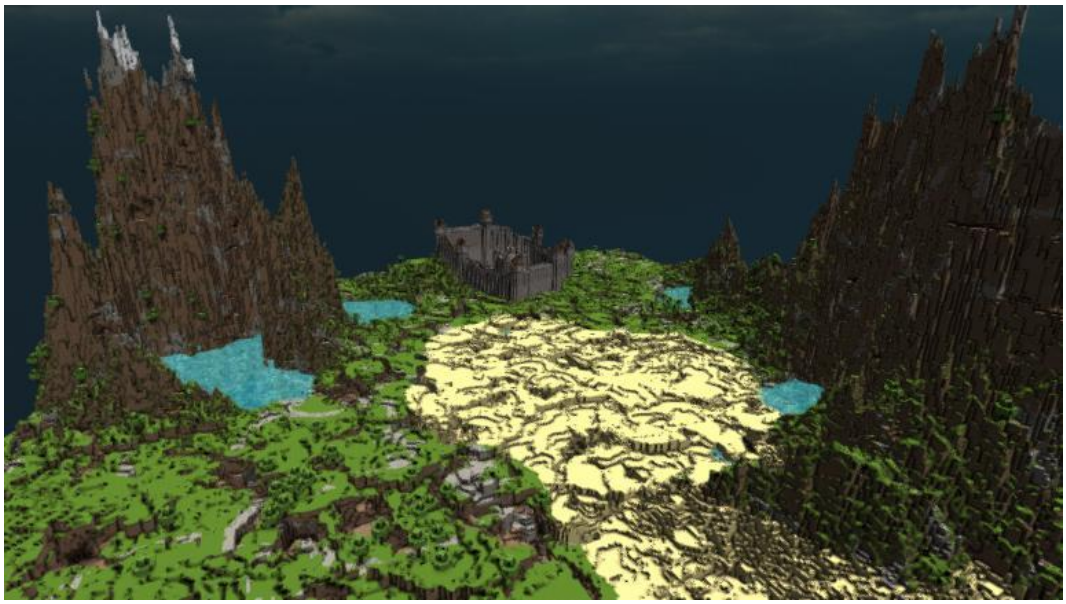
Appendix C

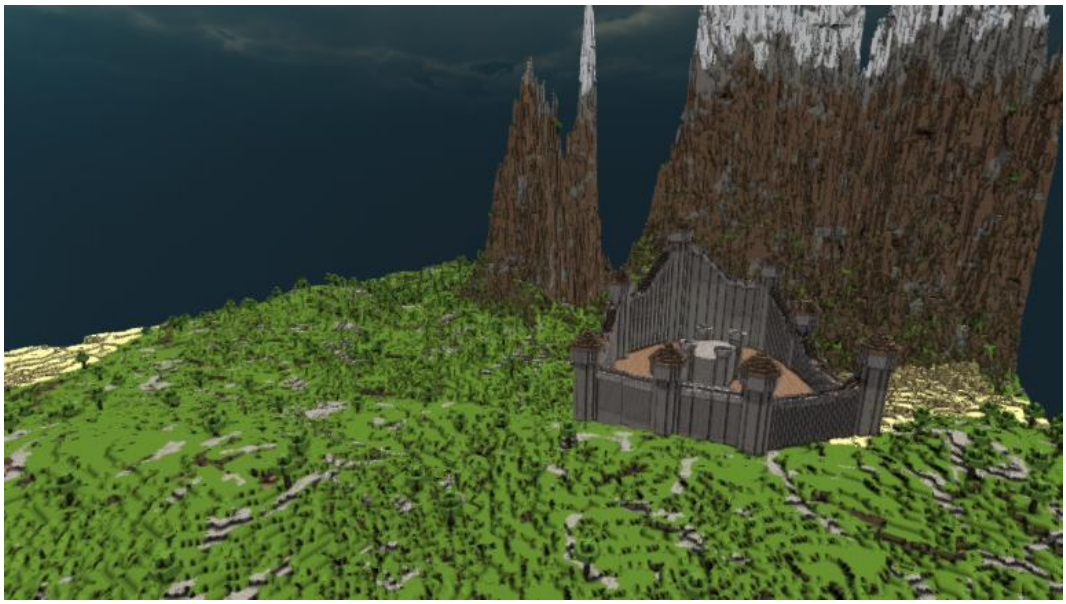
This appendix shows a selection of castles generated for the user study. This should serve as a good showcase of the range of possible outputs of both the terrain generator and the castle generator.

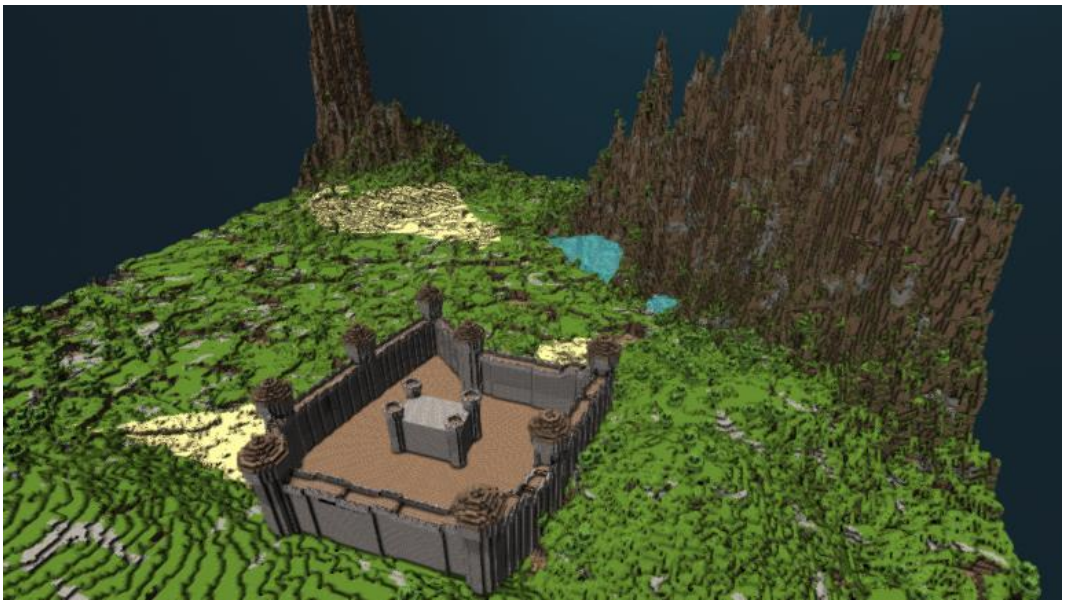


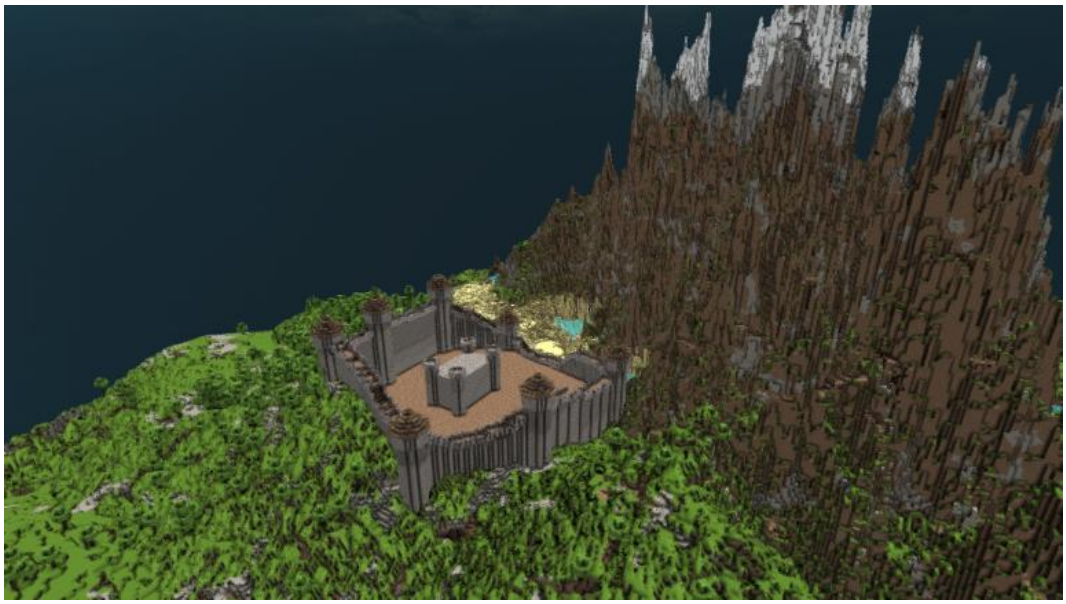


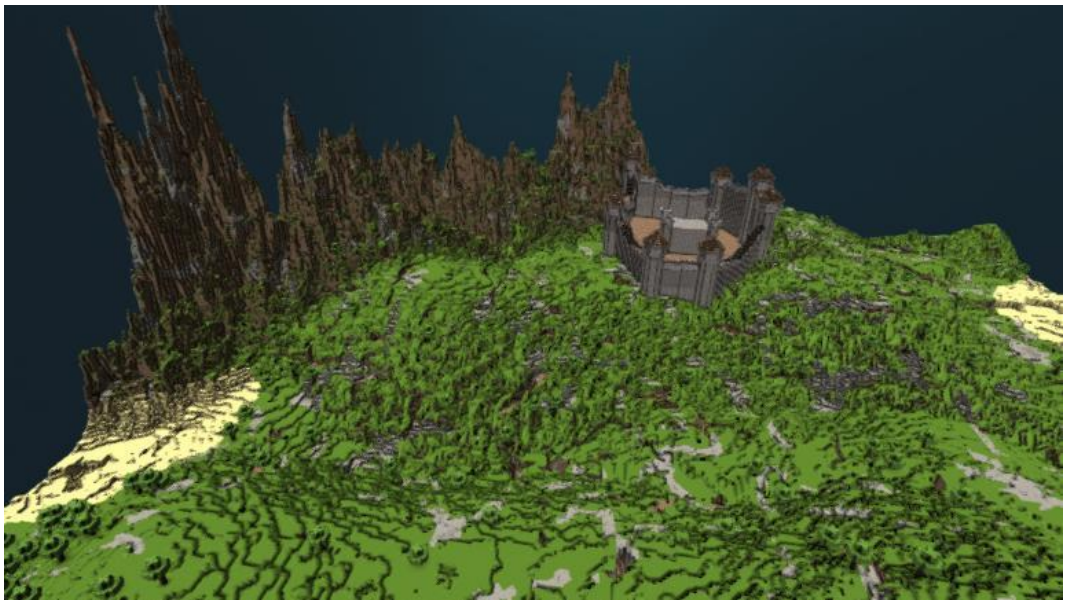


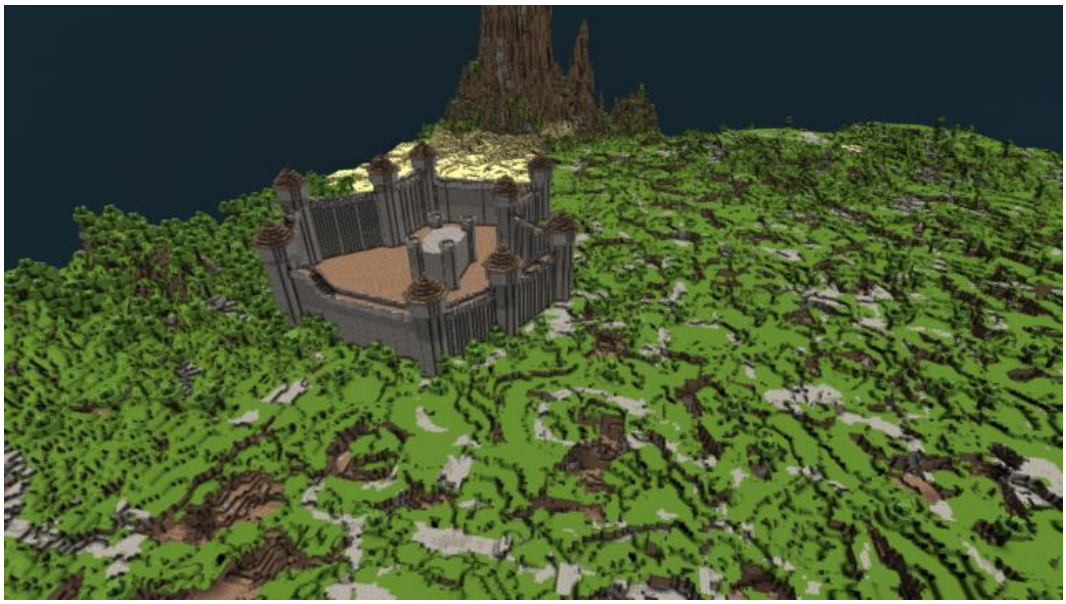


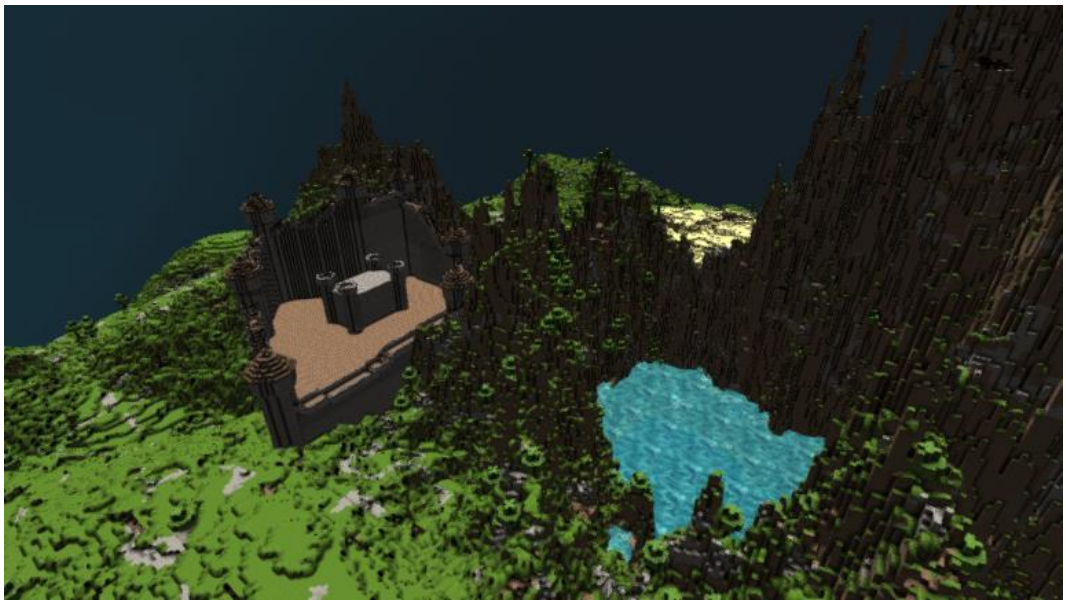


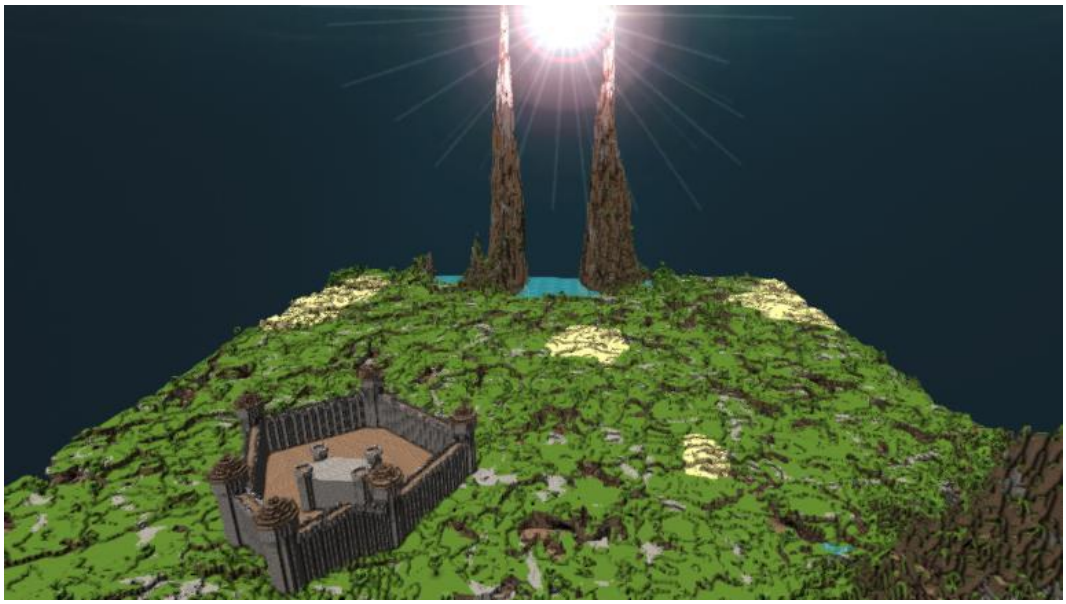


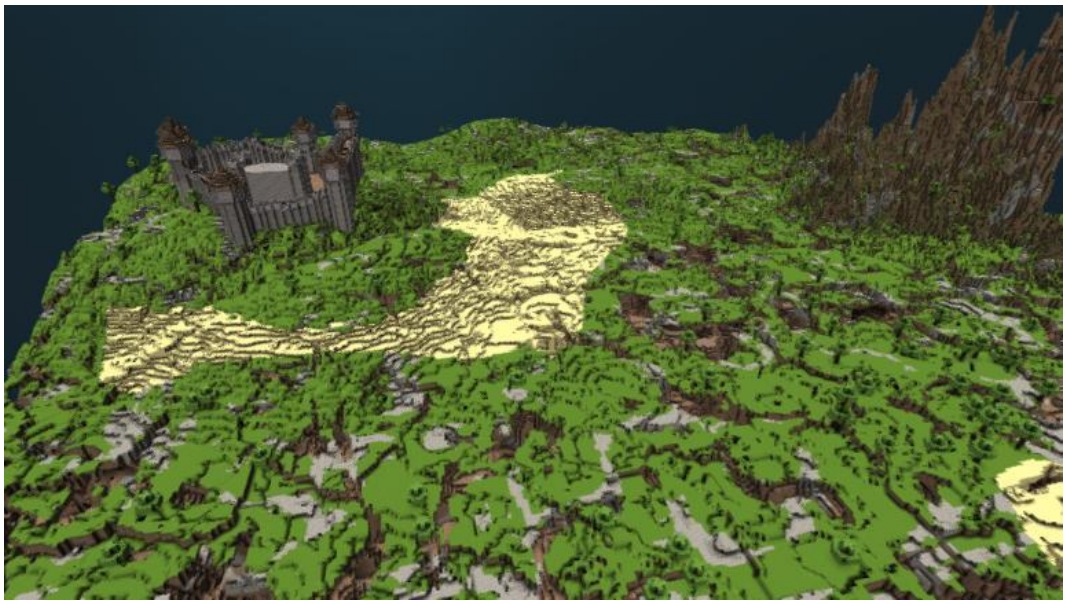


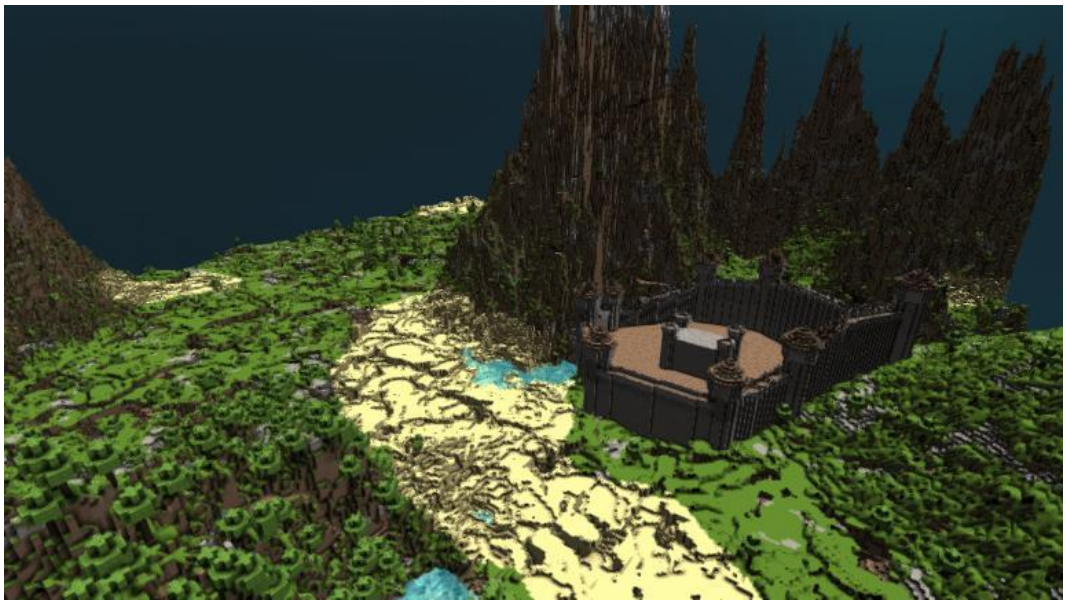


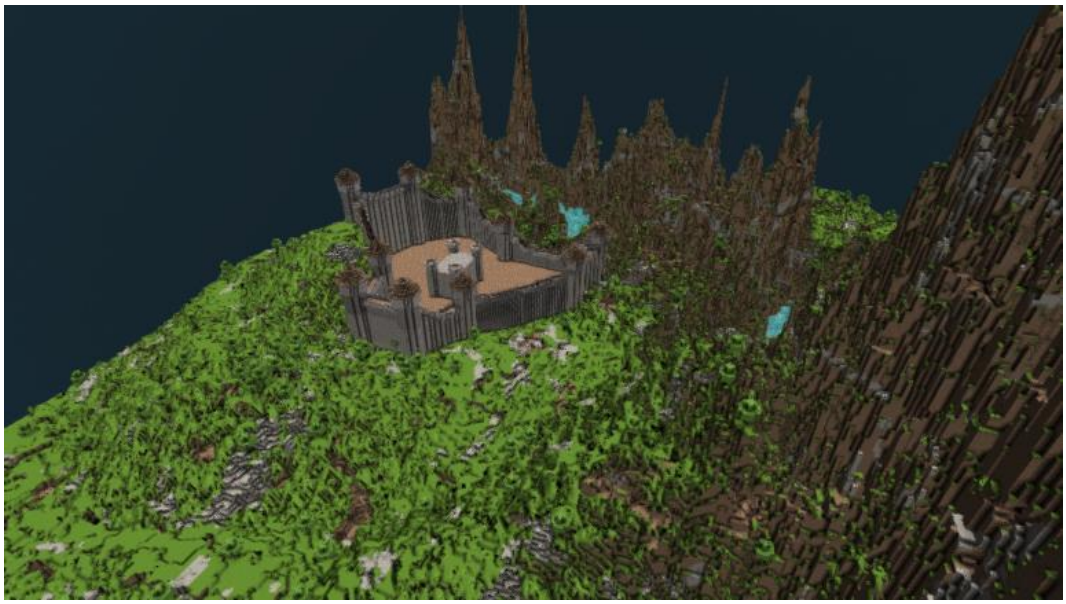


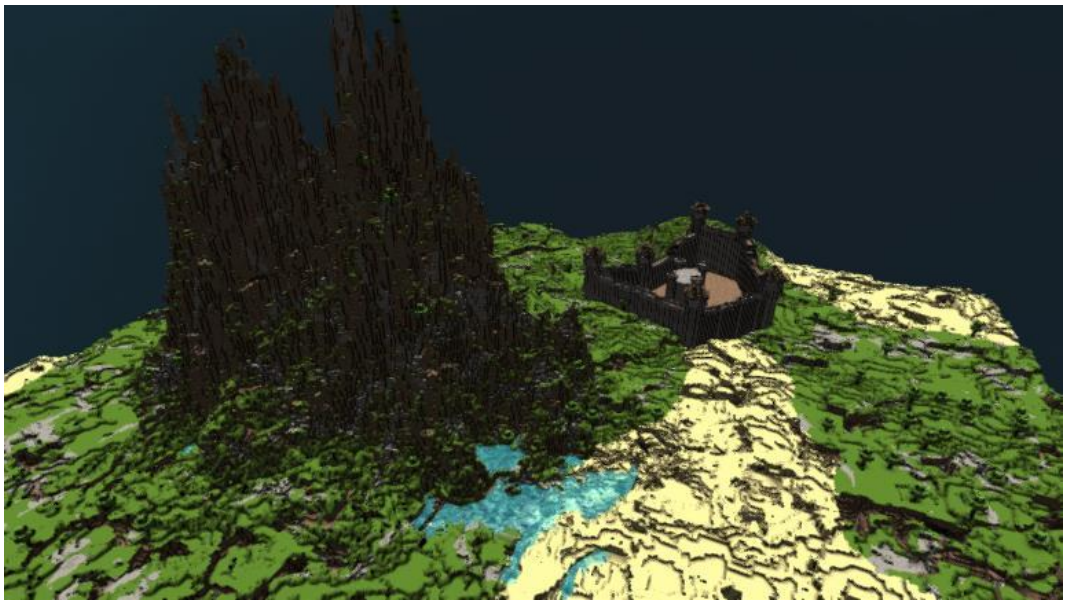


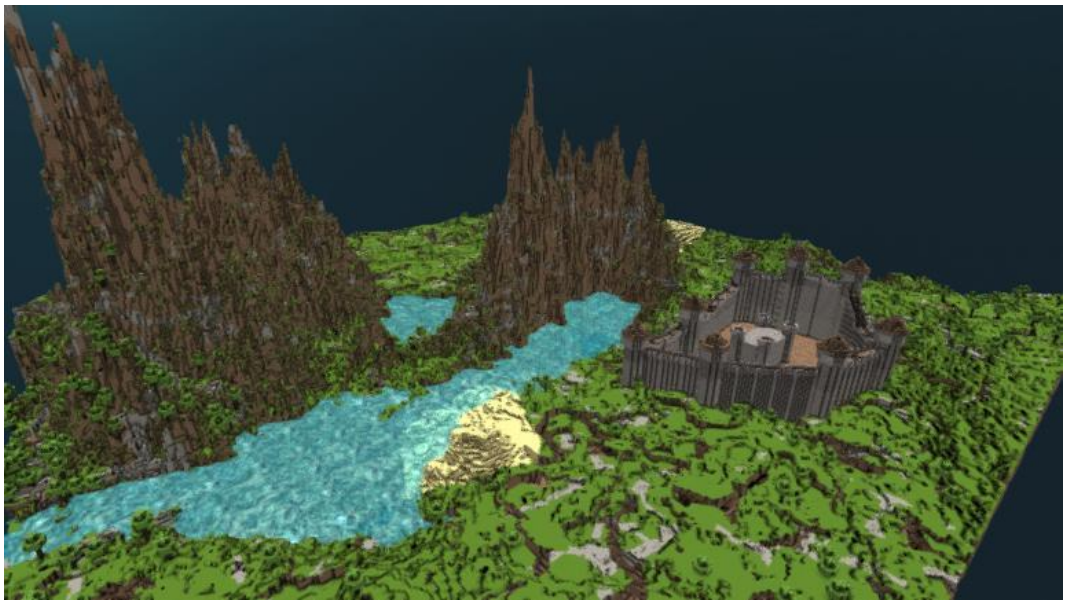


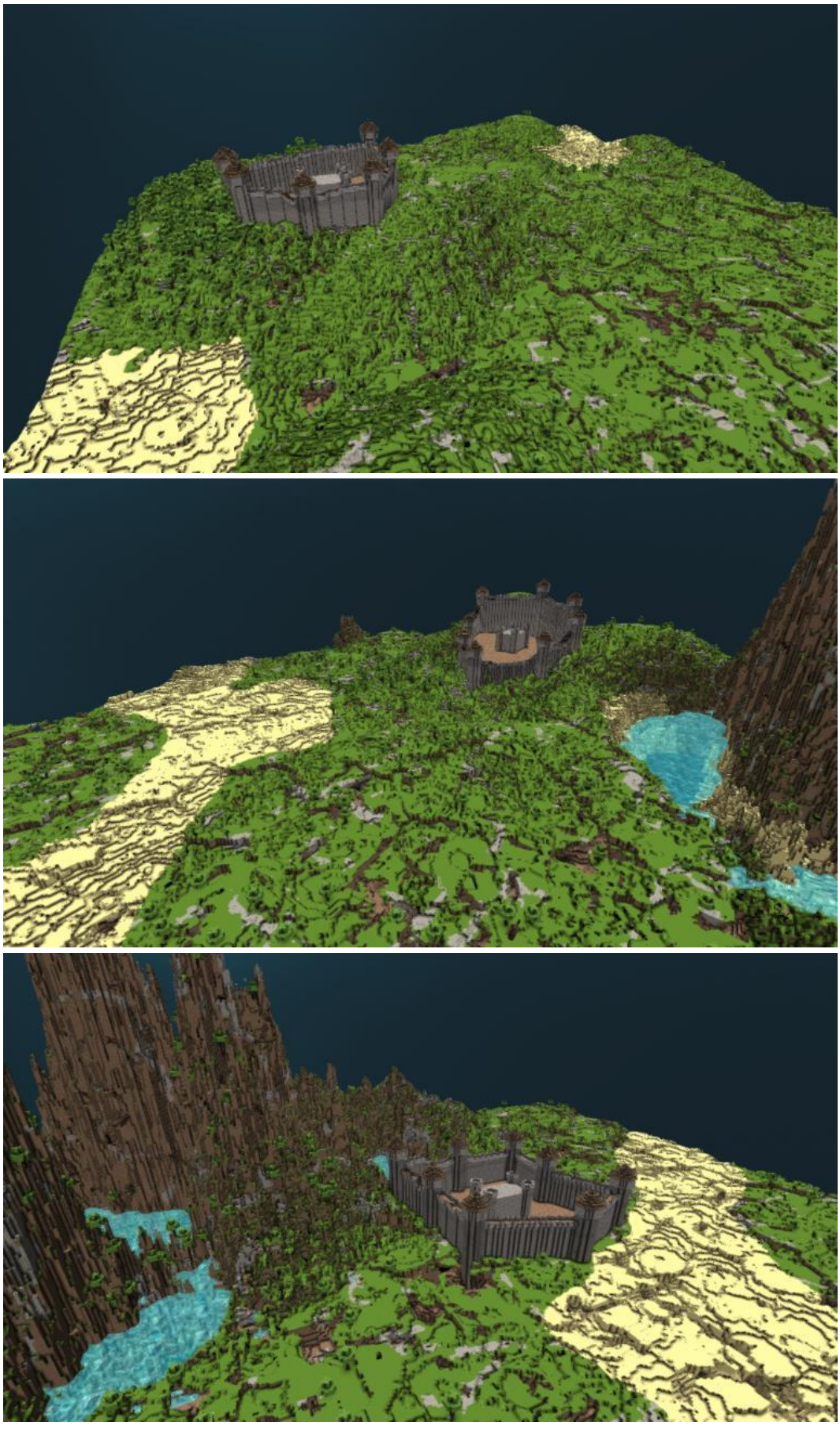












Appendix D

Computing and Mathematical Sciences
Rorohiko me ngā Pūtaiao Pāngarau
The University of Waikato
Private Bag 3105
Hamilton
New Zealand

Phone +64 7 838 4021
www.fcms.waikato.ac.nz

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

15 June 2015

Sebastian Dusterwald,
C/- Department of Computer Science
THE UNIVERSITY OF WAIKATO

Dear Sebastian

Request for approval to conduct a user study with human participants

I have considered your request to conduct an on-line survey for your MSc research project "Procedural Generation of Castles in Voxel Worlds" which investigates a method for automatically generating castles for voxels world games.

You will be using Qualtrix, which is the University of Waikato's recommended on-line survey software.

You state that no participants will be named in any reports or publications.

The research participants' information sheet, and survey method meet the requirements of the University's human research ethics policies and procedures.

Yours sincerely,

Nic Vanderschantz
Human Research Ethics Committee
School of Computing and Mathematical Sciences