UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Märt Mäemees

Dwarf Block Game Development - Dynamic Environment

Bachelor's Thesis (9 ECTS)

Supervisor: Jaanus Jaggo, MSc

Tartu 2020

Dwarf Block Game Development - Dynamic Environment

Abstract:
This Bachelor's thesis describes the techniques used to implement a dynamic voxel terrain system for the game Dwarf Block. It is explained how the marching cubes algorithm was used to build the meshes in Unreal Engine 4. Performance of the mesh building is measured and used to validate that the system is suitable for use in a realtime game. For procedural generation of the landscape, three different terrain data generator implementations are compared. From these, the best one is determined based on opinions gathered through user testing.

Keywords:
Voxel, marching cubes, Unreal Engine 4, computer graphics, procedural generation, dynamic terrain, optimization

CERCS: P170 Computer science, numerical analysis, systems, control

Dwarf Block Mängu Arendus - Dünaamiline Keskkond

Lühikokkuvõte:
Käesolev bakalaureusetöö kirjeldab mängule Dwarf Block dünaamilise vokselmaastiku süsteemi loomiseks kasutatud meetodeid. Töös kirjeldatakse kolmemõõtmelise maastiku loomist, kasutades marssivate kuubikute algoritmi ning implementeeritakse see Unreal Engine 4 mootoris. Samuti kontrollitakse, et võrestiku ehitamise jõudlus sobiks reaalajas mängus kasutamiseks. Maastiku andmete protseduuriliseks genereerimiseks võrreldakse kolme generaatori implementatsiooni ning valitakse neist sobivaim, põhinedes uuringust kogutud andmetele.

Võtmesõnad:
Voksel, marssivad kuubikud, Unreal Engine 4, arvutigraafika, protseduuriline genereerimine, dünaamiline maastik, optimiseerimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Intricate level design can be something that turns a game with good mechanics into a masterpiece. This, however, requires the design team to put a lot of hours into building and designing every level of the game. For games that strive for replay value by offering a different game world every playthrough or are just building a game with a massive world, this is not a sustainable process. Developers who choose to not spend much development time on each level, often go the way of procedural generation. They use algorithms utilizing pseudorandom number generation to generate levels at runtime.

This means that the painstaking work of developers designing each level by hand is now handled by computers and the designers can focus more on other tasks. It does, however, also mean, that the creativity put into the levels by each individual designer is now gone. Therefore, a lot of work needs to go into said algorithms, to give the generated levels a high enough level of detail and variety, so that the players will not notice obvious repetitions and will stay interested throughout a long gameplay loop.

Dwarf Block is an in-development cooperative first-person action game, being built on Unreal Engine 4, where the player and computer controlled characters can move around the world in real time breaking or building parts of the terrain. The objective of each level is to gather enough resources from the world while defending your group of dwarves from waves of enemies and escaping the area with as much loot as possible. The resources can be gathered by mining the earth, necessitating the ability to have dynamic caves and mineshafts that are all destructible.

The goal of this thesis is to make a dynamic voxel terrain system for Dwarf Block. Already existing voxel terrain solutions for Unreal Engine 4 will be analysed and evaluated for use in the game. A custom procedural voxel terrain implementation will be built. The implementation is divided into two main parts: generation of the terrain data and building a visual representation of the data as a terrain.

At first the methods for voxel terrain visualization will be researched. After that, the gameplay architecture of Unreal Engine 4 and ways to implement procedurally generated meshes in it will be looked into, to integrate the system with the engine. The creation of the visual representation will be handled by utilizing the marching cubes algorithm to extract a triangle mesh out of the voxel data. For texturing the procedural mesh, the triplanar projection method will be used. For generating the terrain data, multiple

generator implementations will be created to compare the results of different methods.

Since the terrain is to be destructible during gameplay, the algorithm for extracting the mesh for a chunk of the world, should be fast enough to be run in realtime during gameplay. The performance of the terrain data generators will not be prioritized that much in this thesis, since they need to run only once for each chunk of the terrain.

Once the implementation of the terrain system is done, a demo application will be built around it for distribution to potential players. These players will be asked to compare the terrain created by the different generators as well as general opinions about the system. This will be done to evaluate which of the generators is the most appealing to players and to potentially detect any problems on different hardware configurations.

This thesis will not tackle challenges with mesh extraction, like generating meshes with differing levels of depth, multithreading, or storing the voxel data in structures like octrees. The voxel data generators will not remove terrain imperfections like floating rocks, should the noise generate such features to the landscape.

The second chapter of the thesis describes the background of voxels and voxel mesh generation as well as compares some existing libraries for voxel mesh generation on Unreal Engine 4. The third chapter gives an overview of the implementation details for the procedural terrain and mesh generation. The fourth chapter describes the process for both the performance testing of mesh building and user testing of the terrain generators. It will analyze the results to determine whether or not the system is suitable for usage in a game.

Some of the terms used in the thesis are described in Appendix I. The user guide for the demo created of the final solution can be found in Appendix II.

2 Voxels and Procedural Generation

Many terrain systems use a heightmap system for defining the base model of the terrain. This uses a two dimensional grid with values that determine the terrains height at that spot. The mesh generation is generally quite fast, since the mesh is just a plane grid of squares, where each vertex is equivalent to one element of data in the heightmap, from which the vertex's vertical position is taken from. Examples of this are Unity3D's built-in Terrain[1] component and Unreal Engine 4's Landscape[2] feature.

However, it has its drawbacks, because we are dealing with a three dimensional world, but can have only one value for every vertical column of space. This means that purely heightmap based terrain cannot support more complex terrain features like cave systems, overhangs and arches. For this we need to have three dimensional data.

2.1 Voxels and Voxel Mesh Generation

A voxel is an element that represents a volume in a three dimensional vector space. Terrain can be defined by dividing the space into a uniform axis-aligned three-dimensional grid of voxels. Each voxel can hold values that describe the terrain in the given volume, for example a value could describe the type of terrain that forms the majority in that space. For our purpose we also need a value that depicts how much of the space is actually taken up by terrain. This value will allow deciding where the surface between ground and air, that has to be rendered, lies.

Using voxels, a very detailed terrain can be built, using small building blocks. This allows running more advanced simulations on the terrain data, using more intricate world generation algorithms that no longer have to just define height of the terrain and allows altering the terrain based on user input with much more freedom. This freedom does come at a cost. Storing this data in a three-dimensional grid makes the memory consumption significantly higher than the two-dimensional solutions.

Another aspect that gains significantly more complexity is rendering. Greeff [Gre09] has written about some of the techniques that can be used to render volumetric voxel data:

[1] https://docs.unity3d.com/Manual/terrain-Heightmaps.html

[2] https://docs.unrealengine.com/en-US/Engine/Landscape/TechnicalGuide/index.html

1. Volume ray casting can be used to raycast every pixel to figure out which voxel is being hit.

2. Splatting can be used to backwards project every voxel onto a plane. These projected voxels on the plane are then combined to form a rendered image.

3. Mesh extraction allows taking the set of voxels and processing it into a mesh made of triangles. This mesh can be rendered using the standard techniques that most graphics cards have been built for.

They also talk about the marching cubes voxel mesh extraction algorithm, where changes to a single voxel only affect the small surrounding area, meaning that localized edits that don't require re-extracting the whole dataset can be made.

2.1.1 Marching Cubes

The marching cubes algorithm, developed by Lorensen and Cline [LC87], allows creating a three-dimensional polygon mesh out of a grid of voxels. Its authors have described it as an algorithm that examines a cube with a voxel element at each of its corners. For each corner it is determined whether or not the corner is inside or outside of the object. Since there are eight corners of a cube and each can have a binary value, there are in total 256 possible combinations. For each of those combinations, there is a predetermined set of triangles that are rendered inside that cube. For each one of those triangles, the three triangle corners are all on separate edges of the cube, meaning that the set of triangles can be defined by an array of cube edge indexes. Once the triangles for one cube are constructed, the algorithm moves to the next cube.

The original Lorensen and Cline [LC87] paper introduced 15 triangulated equivalence classes (Figure 1), from which the other 241 cases could be permuted, using two symmetries. When the corner voxel values of the cube are inverted, the triangle structure inside the cube stays the same, the triangles just have to be flipped around. For example, when inverting the voxel values of equivalence class #11 (Figure 1), we get a cube that has its top voxels inside the surface and the bottom voxels outside, meaning that the surface stays in the same position, but should face downward. This halves the required triangulations. Rest of the combinations can be reduced to the 15 equivalence classes by using the rotational symmetry property of the cubes, meaning that if the

given combination can be rotated to achieve one of the equivalence classes, then the required triangulated mesh can be achieved by reversing that rotation on the mesh of the equivalence class.

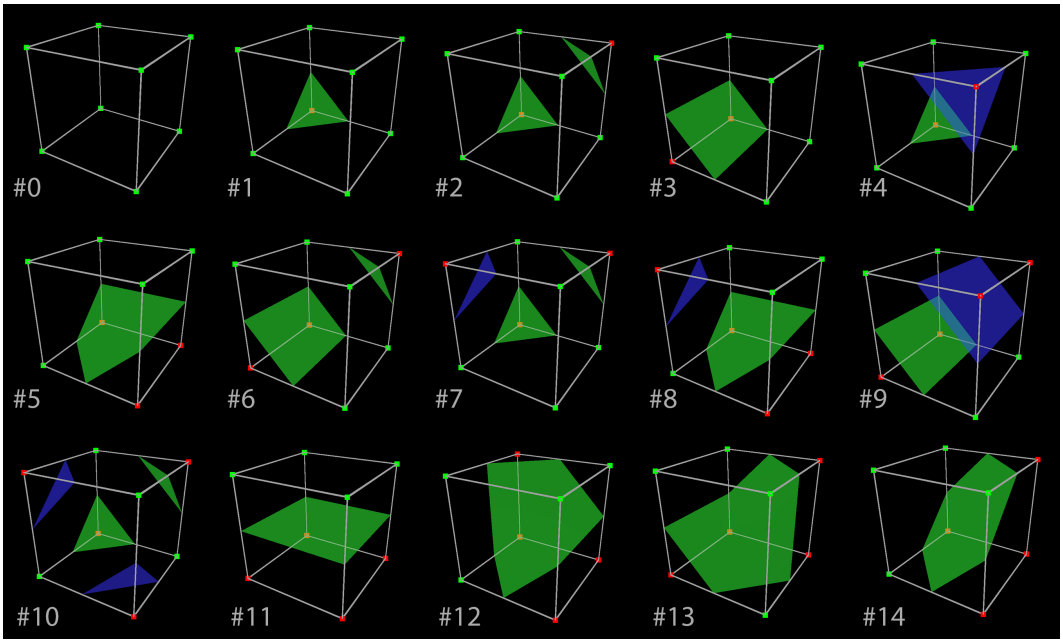

Figure 1. 15 equivalence classes of the marching cubes algorithm. A red dot represents a voxel inside the surface, while a green dot represents a voxel outside the surface. Front-facing triangles are colored green and back-facing triangles are colored blue.

The ability to predefine the triangles for all combinations gives the marching cubes algorithm a significant performance benefit, since no complex triangulation has to be performed every cycle. Instead, the 256 possible combinations are used to create an index of lists. These lists hold cube edges, that should be connected to form the triangles required for the mesh.

Since the marching cubes algorithm only views the voxels in the eight corners of the cube, edits to one voxel in a data set only affect the mesh structure of the eight cubes that surround the voxel. This means that edits to the voxels can be made without requiring the meshing of the whole terrain again.

2.2 Existing Solutions

Since voxel terrain is a feature that many games have been going for, there are multiple libraries that already offer voxel terrain generation. Two prevalent plugins for Unreal Engine 4, the game engine choice for this project, were examined.

2.2.1 Voxel Farm

Voxel Farm[3] is a procedural voxel engine that has been used by major gaming companies like Electronic Arts and Daybreak. It includes a standalone application Voxel Studio, which allows procedurally generating, editing and exporting voxel objects. These objects can be exported into commonly used model formats, or into formats that are used by the Voxel Farm integration for Unreal Engine 4 and Unity3D.

This software is not only used for video games, but also for general data visualization. This includes visualizing real-life locations from massive datasets, while not only focusing on the object surface, but also the insides.

This integration, however, comes at a hefty price, with the cheapest option providing you with precompiled binaries that target only Windows. For multiplatform support you would need the PRO license and full source code access would require negotiating a TRIPLE-A license. This makes this library not suitable for Dwarf Block, due to its large price point.

2.2.2 Voxel Plugin

Voxel Plugin[4] is a plugin made specifically for Unreal Engine 4. It allows generating and modifying procedural voxel terrain inside the Unreal Engine editor and during runtime.

The library has a free version that is open source under the MIT license. However, it also has a Pro version that adds more advanced multiplayer support, voxel graphs, voxel spawners, voxel physics and importers. The voxel graph feature allows defining world generation logic through the use of the Unreal Engine blueprint syntax. Voxel spawners allow spawning custom foliage and actors in the generated world. The voxel physics feature enables physics on voxel objects that are not connected to the ground plane.

[3] https://www.voxelfarm.com/index.html

[4] https://voxelplugin.com/

The free version of Voxel Plugin does not have the voxel graph functionality, but it does allow implementing custom world generators in C++. The plugin comes with example levels, that demonstrate how to use the different features (Figure 2). Some of these include the paid features. Therefore, they are not usable with the free version as-is. Playtesting the plugin on the included example levels, it was noticeable, that sometimes when chunks changed their level of depth, holes temporarily appeared between the meshes. Due to some features, for example voxel spawners, being locked behind a paywall this plugin is not suitable for the project.

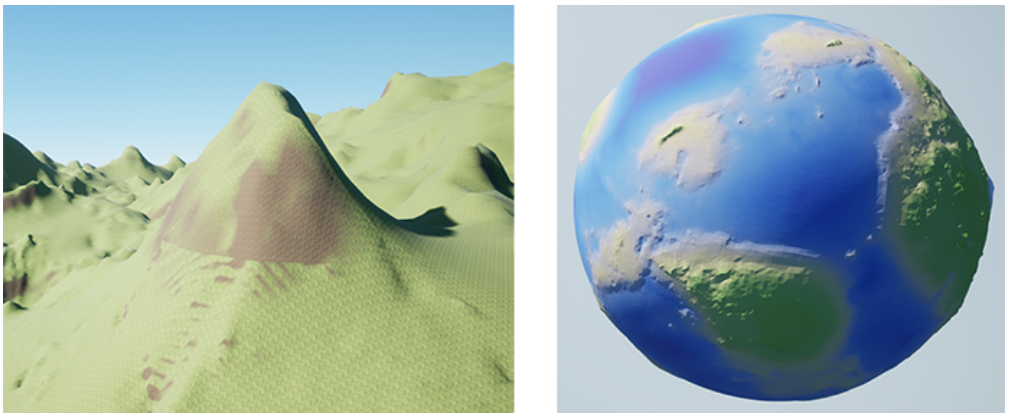

Figure 2. Voxel Plugin example levels Cave_Map (left) and Planet_Map (right).

3 Implementation

Unreal Engine 4 was chosen as the engine for Dwarf Block. This means that the voxel terrain systems need to integrate with the gameplay architecture of the engine. The generated mesh data needs to be usable by Unreal's procedural mesh component and its texturing needs to be done with the Material system.

This section will first discuss the architecture of the system and the way it was integrated into Unreal Engine 4. Section 3.2 describes the implementation of the mesh generation with marching cubes in detail. The three different implementations of the terrain data generators will then be covered. The section ends with a description of why triplanar projection is needed for texturing the mesh and how it works.

3.1 Architecture

Unreal Engine 4 defines a relatively rigid architecture for programming gameplay systems. Unreal games are run inside levels, which contain a hierarchy of actors. Actors are objects that are placable inside levels, whether by the developers hand in the editor or programmatically during runtime. The engine defines an interface of functions, that are called at certain points in the actors lifecycle. These can be implemented to define the actor's behavior during gameplay.

Smaller functionalities can be separated into reusable actor components, which are very similar to actors, but instead of being created in the level hierarchy, they are placed inside an actor's internal hierarchy. This allows compositing the same functionality to different types of actors.

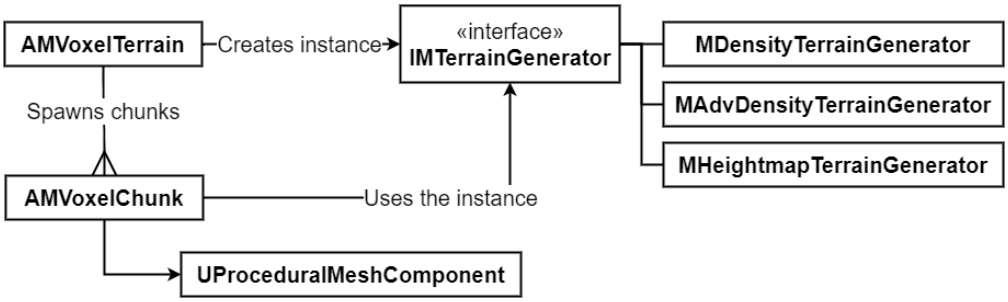

Figure 3. Diagram of the classes that form the core of the terrain functionality. Classes that are implementations of actors are prefixed with 'A'.

The solution implements a root terrain actor AMVoxelTerrain (Figure 3), that manages the world terrain and defines the shared parameters used for terrain generation. This terrain actor doesn't generate any mesh itself, but it spawns a grid of chunk actors (AMVoxelChunk), which handle the terrain mesh generation within their area. These chunks contain an Unreal's procedural mesh component, which allows creating a mesh during runtime programmatically, by providing arrays of vertices, triangle indices, normals, etc.

The chunks need to get their voxel values from somewhere, that ensures that the data is consistent with surrounding chunks. For this, the terrain actor instantiates an implementation of the terrain generator interface IMTerrainGenerator (Figure 3) and provides it to the chunk when loading. This generator implements a function, that given the world position of the chunks lowest corner (smallest coordinates on all axes), returns an array of voxel data for the whole chunk. There are multiple implementations for the generator that can be chosen by the developer in the terrain actors parameters.

3.2 Mesh Generation

The terrain is divided into cubic chunks that have a width n that is a power of two. The voxel values are also not kept in a singular container, but are separated into the chunks that contain them. This means that each chunk can be individually loaded and unloaded. Each chunk contains a one-dimensional array of structs that hold the voxel data. The struct contains a `uint8` field that defines the density value, which is used to determine, whether or not the voxel is inside or outside the surface.

When building the mesh of a chunk, the chunk is divided into n^3 cubes, which have voxel values in their corners (Figure 4), as described in Section 2.1.1. These cubes are traversed sequentially, starting from the cube with the lowest coordinates in the chunk. For each cube, a cube index is calculated. This index defines which of the 256 possible combinations the cube represents.

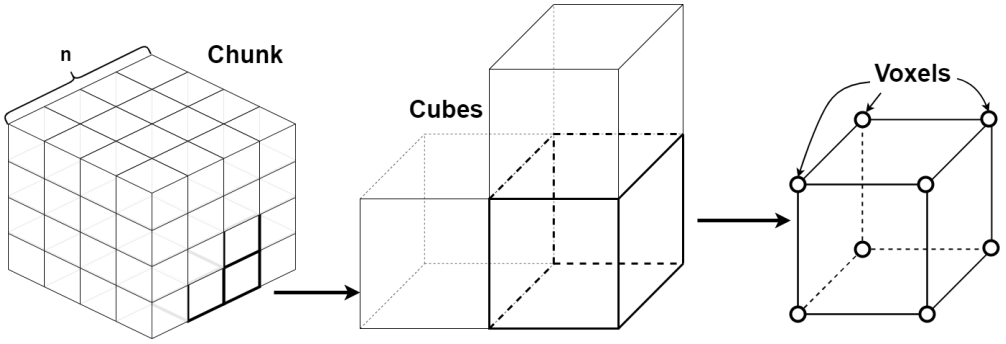

Figure 4. Visualization of the chunk structure.

The cube index is calculated, by checking each corner of the cube and if the voxel in that corner is determined to be inside the surface, the bit in the cube index corresponding to that corner (see Figure 5) is turned positive. A voxel is considered to be inside the surface, if its density value is higher than the constant density threshold (Figure 6). For example, if the corners 2 and 6 are inside the surface, the resulting cube index would be $01000100 = 68$.

If all eight bits of the cube index are set to the same value, then the whole cube is either inside or outside the surface and the surface doesn't intersect with the cube. This means that the mesh extraction algorithm can immediately move on to the next cube (Figure 6). Otherwise, the algorithm will continue with creating the triangles that belong inside the cube.

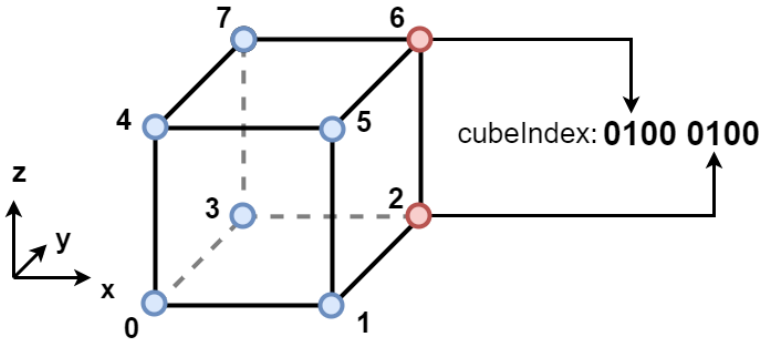

Figure 5. Each corner of the cube gets assigned an index. These indices are used to calculate the cube index for the cube. Red corners are considered to be inside the surface.

```
int32 cubeIndex = 0;
if (GetTerrainValue(0, pos) > DENSITY_THRESHOLD) cubeIndex |= 1;
if (GetTerrainValue(1, pos) > DENSITY_THRESHOLD) cubeIndex |= 2;
if (GetTerrainValue(2, pos) > DENSITY_THRESHOLD) cubeIndex |= 4;
if (GetTerrainValue(3, pos) > DENSITY_THRESHOLD) cubeIndex |= 8;
if (GetTerrainValue(4, pos) > DENSITY_THRESHOLD) cubeIndex |= 16;
if (GetTerrainValue(5, pos) > DENSITY_THRESHOLD) cubeIndex |= 32;
if (GetTerrainValue(6, pos) > DENSITY_THRESHOLD) cubeIndex |= 64;
if (GetTerrainValue(7, pos) > DENSITY_THRESHOLD) cubeIndex |= 128;

// Cube does not contain any triangles.
if (cubeIndex == 0 || cubeIndex == 255) {
    continue; // Move to the next cube.
}
```

Figure 6. Calculating the cube index. DENSITY_THRESHOLD is the threshold above which the voxel is considered to be inside the surface. GetTerrainValue takes in the index of a corner and the local position of the cube inside the chunk (based on the corner with index 0) and returns the density value of the voxel in that corner.

Using the cube index, a list of edges can be looked up from the triangle table. Only a list of edges is needed, because the triangle vertices can only be placed on cube edges. This edge list contains 16 integers, with the last integer always being -1. The first 15

integers form up to 5 triplets, which represent the triangles to be constructed. Each integer of a triplet specifies an edge that one of the corners of the triangle should reside on. The order of the integers inside the triplet defines which way the triangle will face. In this case, clockwise direction results in a front-facing triangle. These triplets are iterated through until a -1 value is encountered. This means that there are no more triangles and the algorithm can proceed. The cube index 68 from the last example, would create triangles as depicted on Figure 7.

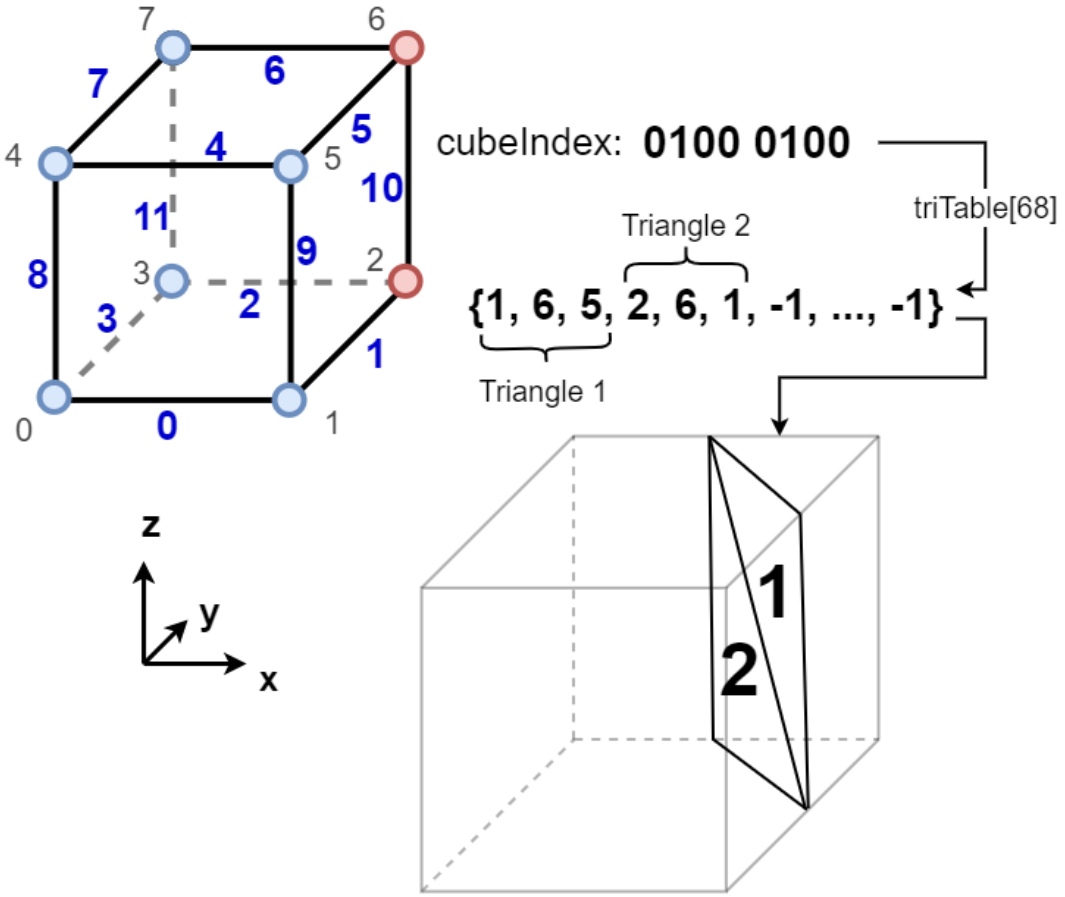

Figure 7. Constructing triangles from a cube index. Edge indexes are marked with blue numbers.

Every chunk has n^3 cubes that are rendered by it, but they also own the data of only n^3 voxels. To extract the mesh for n^3 uniform cubes, with a voxel at each corner of a

cube, $(n+1)^3$ voxels are needed. This means, that for cubes that reside on the edges where at least one coordinate is highest within chunk local space, voxels from adjacent chunks need to be viewed (Figure 8). Therefore, if the chunks are all placed together in a rectangular pattern, as is the case here, there must be an extra layer of chunks, that have their voxels loaded but are not rendered, in the positive direction of all three axes.

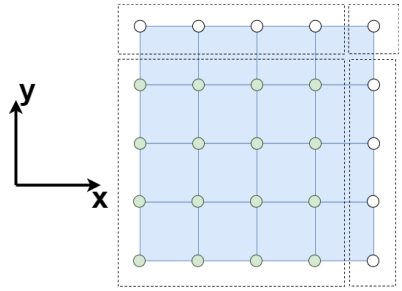

Figure 8. Top-down visualization of the cubes contained within one chunk ($n = 4$) and the voxels required to build the mesh for them. Green dots are voxels owned by the chunk, white dots are owned by adjacent chunks (different chunks shown by dotted boundaries).

The position of each vertex on an edge is determined through interpolation. The solution is based off of Cory Bloyd's example implementation [Blo]. This interpolation is done between the two endpoints of the cube. The weight for the interpolation is found by calculating the difference between the density threshold and the first corner's voxel value. This difference is then divided with the difference between the voxel values of both corners, as can be seen on Figure 9.

```
1 // value field is the density value of the corresponding voxel.
2 float weight = (float) (DENSITY_THRESHOLD - voxA.value) / (float) (
    voxB.value - voxA.value);
3 FVector pos = posA + (posB - posA) * weight;
```

Figure 9. Finding the position of a vertex on a cube edge.

The cubes of the chunk can be built completely independently of other cubes, but the resulting mesh would be flat shaded, with each triangle producing its own vertices and

normals. To build smoother meshes that work well with the chosen texturing method, the vertices produced by the cubes at the same locations are merged together with the following algorithm:

1. For every triangle in a cube a normal vector is calculated to be perpendicular with the triangle.

2. Every vertex of a triangle is placed into a map along with a list with its generated normal. If the vertex is on an edge, it is mapped by the calculated edge index (Figure 10). This can be done, because all four cubes that contain the edge, would create the vertex on that edge in the same position. If the vertex is in a corner of the cube, it is mapped into a separate map by the calculated corner index (Figure 11).

3. Once all the cubes are iterated through, the maps with the vertices are iterated through. For each vertex the list of normals is used to find an average normal vector, which is added to the mesh with the vertex position.

These operations produce a mesh that doesn't have duplicate vertices and isn't flat shaded. The resulting mesh can be more impressive visually, when paired with a suitable material and is significantly more performant due to the lower vertex count.

This method works for smoothing the mesh within a chunk, but chunk borders can still remain unsmoothed. To ensure that the chunk borders have no jagged transitions, the algorithm must iterate through another layer of cubes in all directions. In those new cubes, however, building the whole mesh is not necessary. The algorithm has to just calculate the normals for triangles that have vertices on edges or corners that are in the rendered part of the chunk and add the normals to the corresponding list. This means that some triangles are calculated multiple times by multiple chunks, but it guarantees that building the meshes of the chunks is an independent operation. The difference between a flat shaded mesh and a smoothed mesh can be seen on Figure 12.

```
// Account for the extra cubes being iterated through.
int32 xLength = CHUNK_WIDTH + 3;
// Each cube is considered to have 2 layers (horizontal and vertical)
// on the y direction.
int32 yLength = 2 * xLength * xLength;
localPosition.Y *= 2;
localPosition.Z *= 2;
FIntVector pos = localPosition + edgeIndexingOffset[cubeEdge];
// Turn three-dimensional coordinates into a single index.
return pos.X + pos.Y * xLength + pos.Z * yLength;
```

Figure 10. Calculating the edge index.

```
FIntVector pos = localPosition + vertexPositionOffsetInt[cubeVertex];
// Account for the extra cubes being iterated through.
int32 w = CHUNK_WIDTH + 3;
// Turn three-dimensional coordinates into a single index.
return pos.X + pos.Y * w + pos.Z * w * w;
```

Figure 11. Calculating the corner index.

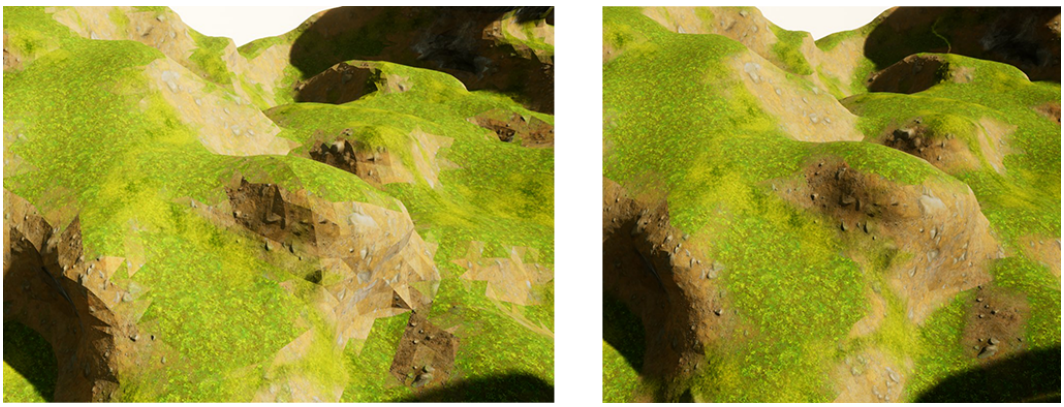

Figure 12. Mesh with individual vertices (left). Smoothed mesh with merged vertices (right).

3.3 Procedural Voxel Data Generation

The voxel data for the terrain is generated on a per-chunk basis. The implementations of the terrain generator interface (Section 3.1) implement a function that takes in a chunk's world position and generates an array filled with the chunks voxel data, which is returned. All of the generators implemented in the scope of this thesis use noise functions in different ways to achieve unique terrain that has no size limits and requires very little input from the developers.

3.3.1 Density Terrain Generator

The density terrain generator iterates over all three dimensions and at every voxel point calculates a single value from a simplex noise function that accepts three input coordinates (Figure 13). Since the noise value can be negative, the value is brought into a positive range, by adding one to it. The noise value is then multiplied by a factor, which determines how much the noise affects the terrain. The resulting value is substracted from a base value of 255.0, which is the maximum density value.

```
1 pos.Z *= Parameters.NoiseVerticalScale;
2 float density = 255.0f;
3 density -= (NoiseGenerator.GetNoise(pos.X, pos.Y, pos.Z) + 1.0f) *
      Parameters.NoiseMultiplier;
4 density -= pos.Z * Parameters.FieldVerticalScale;
```

Figure 13. Calculating the density at a certain position. The NoiseVerticalScale parameter can be used to adjust the noise frequency on the z-axis.

The generated mesh, however, does not yet look like realistic terrain. It looks like random noise going infinitely in all three dimensions, as can be seen on Figure 14. Terrain should have a ground plane, above which no more regular terrain is generated. To achieve this, the world position on the z-axis is multiplied by a factor and substracted from the density value. This results in a decrease in the average density value, the higher up the voxels are, until even the lowest noise value cannot result in the voxels being above the density threshold. This results in the generated mesh being mostly a single ground plane.

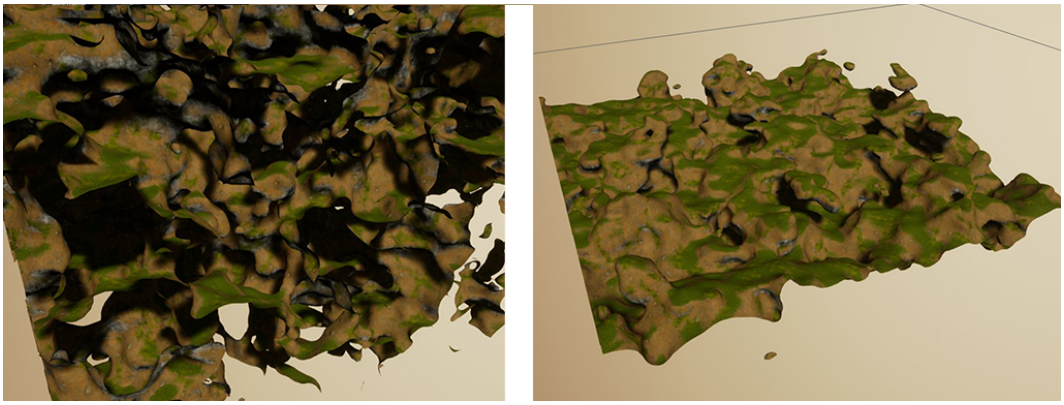

Figure 14. Terrain generated with the Density Terrain Generator without (left) and with (right) the z-axis world position substraction.

3.3.2 Advanced Density Terrain Generator

The advanced density terrain generator iterates on the idea of the basic density terrain generator described in Section 3.3.1. The algorithm initializes the density value at the configured base density value, from which the vertical position multiplied by a factor is then substracted. After that, a technique described by Ryan Geiss [Gei07] is used, by utilizing multiple noise generators with differing parameters to generate values that are added to the density. In general, the frequency and amplitude of these noise generators are inversely related. When one is larger, the other is set lower. This generates larger features on the landscape, while layering more frequent detail at lower amplitude on top of it.

By default the output of the noise functions is approximately bounded from -1.0 and 1.0. The output of noise generators, that provide the values with the largest amplitudes, are shifted more towards the positive range (Figure 15). This provides the landscape with large mountains occasionally, while keeping the valleys generated from being too deep.

```
NoiseGeneratorMountain1.SetFrequency(NoiseFrequency * 0.25f);
NoiseGeneratorMountain2.SetFrequency(NoiseFrequency * 0.1f);
// ...
density += (NoiseGenMountain1.GetNoise(pos.X, pos.Y, pos.Z) + 0.50f)
    * NoiseAmplitude * 3.0f;
density += (NoiseGenMountain2.GetNoise(pos.X, pos.Y, pos.Z) + 0.75f)
    * NoiseAmplitude * 7.5f;
```

Figure 15. Two noise values at low frequency, but high amplitude added to the density.

Ryan Geiss [Gei07] has said, that using layered noise like this can result in terrain that is feature rich. But the way detail is created can start feeling too regular. To counter this, he suggests warping the world space coordinates with noise, before the noise calculations for the terrain. The advanced density terrain generator provides an option to warp the world coordinates with noise of given frequency and amplitude (Figure 16).

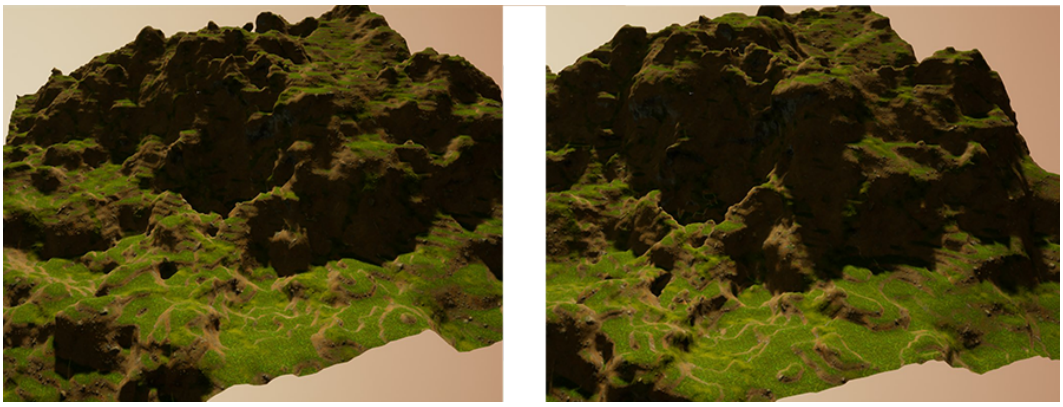

Figure 16. Advanced density terrain generator without (left) and with (right) world coordinate warping.

This method of terrain generation can run into the problem of generating a ground plane below the vertical zero-coordinate. Since no chunks are rendered below that, it would leave holes in the visible terrain. An exaggerated example of this can be seen on Figure 18. To circumvent this, an option for defining a base terrain height is provided. At and below this base height, the density must always be higher than the threshold. A frequency and amplitude can be provided, to offset the base height by small amounts

using a two-dimensional noise function, preventing the plane, generated in places where the base height is needed, from being completely flat. To make the transition between the terrain from the noise and the base height smoother, a base terrain height amplitude is set. This amplitude defines how high from the base height, the functionality starts to gradually take effect, by adding a fraction of the density threshold to the calculated density (Figure 17). At and below the base height, the added density itself is enough to exceed the density threshold.

```
1 float baseHeightOffset = BaseHeightNoiseGenerator.GetNoise(pos.X, pos
      .Y) * BaseHeightNoiseAmplitude;
2 float baseHeight = BaseTerrainHeight + baseHeightOffset;
3 float baseHeightDiff = pos.Z - baseHeight;
4 density += (DENSITY_THRESHOLD + 1) / BaseTerrainHeightAmplitude
5     * FMath::Max(BaseTerrainHeightAmplitude - baseHeightDiff, 0.0f);
```

Figure 17. A fraction of the density threshold is added to the voxel's density, when close enough to the base height of the terrain.

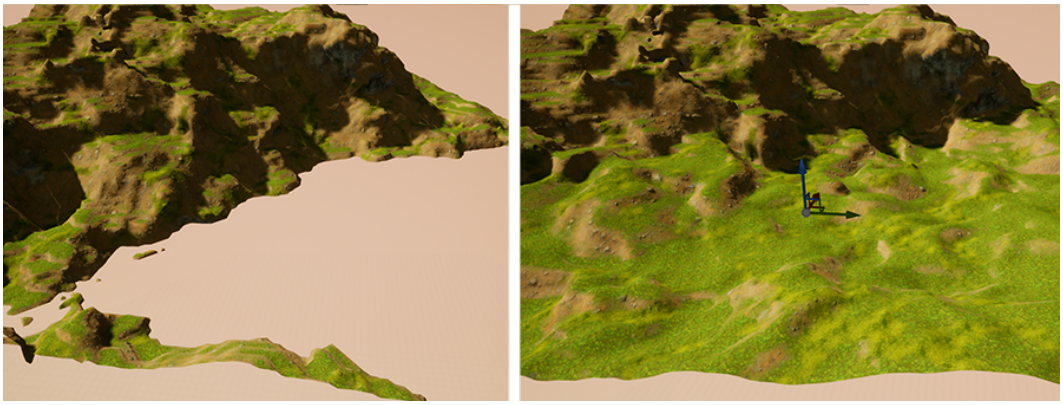

Figure 18. Terrain made by the advanced density terrain generator without the base terrain height (left) and with it (right).

3.3.3 Heightmap Terrain Generator

The heightmap terrain generator makes use of the traditional heightmap technique. It iterates through columns on the xy-plane and calculates the height value for that particular

column from a two-dimensional noise function. Every voxel in that column, that falls below the calculated height value, is then set to the maximum density. The voxel that is on the same height as the calculated column height, the density is set based on the fractional part of the calculated height. The higher, the fractional part is, the higher above the density threshold, the voxel's density is set.

A voxel terrain, however, does not have to be only limited to the two-dimensional heightmap. By using a three-dimensional noise function with different seed and frequency, basic caves can be carved into the landscape. If the calculated noise value is lower than the cave threshold, the voxel at that spot is set with a minimal density value. This forms holes on the ground plane and inside the terrain (Figure 19).

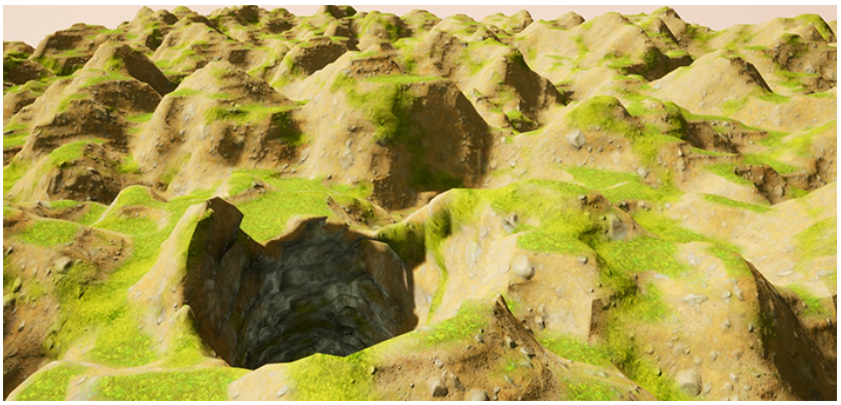

Figure 19. Landscape created by the heightmap terrain generator. A cave intersecting the ground plane can be seen on the left.

3.4 Texturing

A texture is essentially a two-dimensional array of color values. The two-dimensional coordinates of the array have to be somehow mapped to the three-dimensional meshes. Traditionally, when dealing with existing meshes or simple generated meshes, this is done by UV mapping. UV mapping embeds two-dimensional coordinates to every vertex of the mesh which are then interpolated throughout the triangles of the mesh and used to lookup the color value from the texture.

Computing the UV coordinates for an arbitrary mesh as complex as one generated by the marching cubes algorithm is difficult, since the UV coordinates cannot be set

per-cube, taking into account just the cube index, but have to consider the rest of the mesh as well to avoid distortions. In this solution a different approach, that embeds no texture mapping info into the mesh, but performs all calculations during render time in the shaders, is taken.

A tiling texture can be projected to a mesh without predefined texture coordinates, by using the world position of the fragment. For example, by taking the fractional part of the world space coordinates x and y and using them as texture coordinates, planar projection on the xy-plane is achieved. This looks good on horizontal planes, but on more vertical places, the texture is stretched. This is because the x and y coordinates change very little, or even stay the same on completely vertical planes, as seen on the sides of the cube on Figure 20.

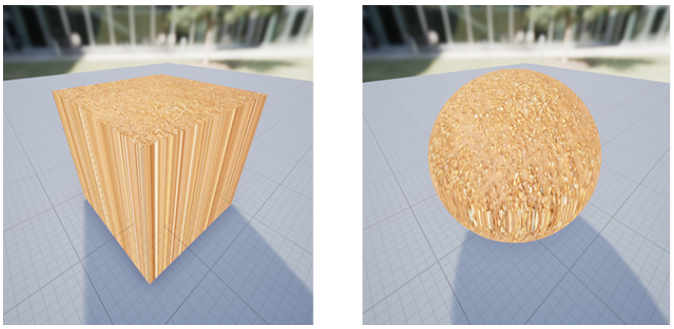

Figure 20. Planar projection on the xy-plane.

Brent Owens [Owe] writes that this problem can be solved, by using three planar projections. Utilizing the interpolated normal at the fragment, the three texture samples from planar projections of all planes along the primary axes can be blended together, to achieve the texture value most suitable in that position. This method is called triplanar projection.

Owens explains that the blending is done, by taking each of the planar projections and multiplying it with the projection of the normal on the primary axis, that isn't on the plane of the planar projection. To get the blending to be from the same sample values, for both positive and negative direction of the axes, the normal vector in this operation is turned positive beforehand. For example, the texture sample taken for the XY-plane projection, is multiplied with the z-component of the absolute value of the normal vector.

These samples are then added together to get the blended texture value, as can be seen on Figure 21.

```
    // n is the normal vector interpolated from the vertex normals.
    n = pow(abs(n), blend_sharpness);
    // Ensure the weights for blending sum up to 1.0.
    n = n / (n.x + n.y + n.z);
    // colXY is the color texture sample from the planar projection
    // on the XY-plane. colXZ and colYZ are analogous.
    blended_color = colXY * n.z + colXZ * n.y + colYZ * n.x;
```

Figure 21. Pseudocode for blending three texture samples together.

For adjusting the sharpness of the blend (Figure 22) between the different samples, the absolute value of the normal vector can be exponentiated before other operations. After this is done, the vector is divided by the sum of its components, to ensure that the sum of its components is exactly one. This is done because the sum of the texture sample multipliers should be one.

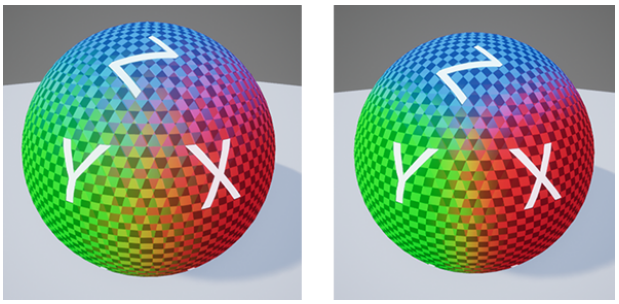

Figure 22. Triplanar texture mapping with blending sharpness of 5 (left) and 10 (right).

Different textures can be sampled for specific planes, to add more variety to the terrain. For example, when sampling for the XY-plane, a texture of grass can be used and when sampling for the other planes a dirt texture can be used, to make the mesh resemble ground (Figure 23). This does place grass on the underside of the mesh as well. To counteract this, a stone texture can be used instead of the grass one, when the z-component of the normal vector is negative.

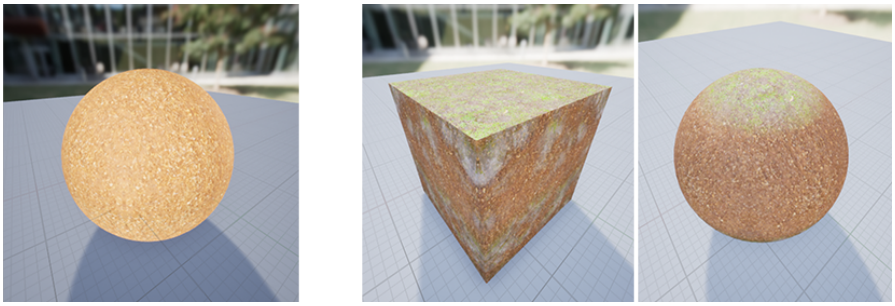

Figure 23. Triplanar projection with a single texture (left) and multiple textures (right).

Triplanar mapping allows texturing a generated mesh without much complexity, but it can be more performance heavy than regular UV mapped texturing, since every fragment requires sampling at least three textures for just the color. This number of samples can grow substantially, when separate textures for heightmaps, normals, etc. are added. Packing some of the textures together, making use of all the color channels in a texture file, would alleviate the problem, but the number of lookups would still be quite high.

4 Testing

The first part of this section will describe the performance testing done to the mesh building functionality. It will analyse the results, find the most performance heavy part and describe the optimization done to alleviate the problem. The performance analysis will determine, whether or not it would be feasible to use this system for building a mesh out of voxel data in the Dwarf Block game.

The second part will describe the user testing done to gather feedback on the different terrain generators developed for the system. Results are analysed to determine the terrain generator that showed the most potential with users.

4.1 Performance Testing

The built system is meant to be used as terrain that allows interaction by either building on it or destroying parts of it during gameplay. Preferrably, if an actor interacts with the terrain, the mesh is updated within the same frame. This means that updating the mesh of a chunk needs to be done fast, to not have a noticeable impact on the framerate. All of the performance tests in this section have been done on the following hardware:

- CPU: Intel Core i5 6500
- GPU: Nvidia GTX 970
- RAM: 16 GB

For profiling performance over time in detail, Unreal Engine 4 provides the `stat startfile` and `stat stopfile`. During the time between these commands, Unreal records the execution time of parts of the engine and saves it into a file. This file can later be opened in the Profiler window to inspect the frame time graph and execution times of individual scopes.

Unreal does allow defining custom scopes, in which execution time and count will be profiled, using its macros. This allows profiling the execution time of the mesh building function specifically. As a testing scenario, a terrain 20 chunks wide and 8 chunks high was built, with one chunk being built each frame, using the default parameters of the advanced terrain generator.

Figure 24 shows a part of the graph detailing the execution time of building the mesh in each frame. The red line represents the whole algorithm for building the mesh, while the yellow line represents part of the algorithm that is done after calculating the cube index. Frames where the yellow line is flat at the zero mark, are frames, where the chunk being built didn't have any surface intersecting it, which means that no mesh needed to be constructed. The area between the red and yellow lines is the time spent on calculating the cube index.

Comparing the blue and yellow lines shows that when the chunk has no surface to render, the execution time of the function stays quite consistent, at slightly above 1ms. This is because the algorithm always performs the same amount of voxel data lookup and comparison operations. The count of operations done by rest of the mesh building function, however, is more variable and depends on how many cubes have a surface intersecting them and how many triangles they generate.

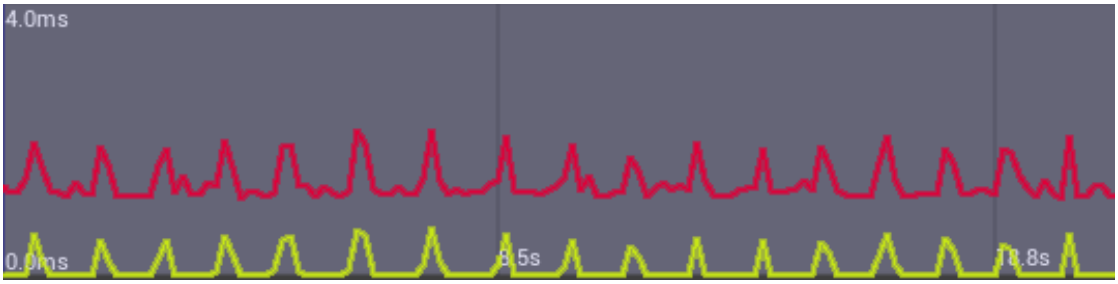

Figure 24. Part of the execution time graph for the mesh building function (red) and the part of the function after cube index calculation (yellow).

Overall, the execution times of chunks, that contain a surface, seems to stay roughly around 2ms. Throughout the whole test, the mean execution time for the mesh building function was 1.297ms. For keeping a 60 frames per second framerate, the frame time should not go above 16.6ms. With the mesh building usually taking around 1 to 3 milliseconds, the terrain system leaves quite a bit of time for other functionalities inside the game loop.

This kind of profiling does incur some performance overhead, meaning the results will not be exactly the same as when running a build of the project. This is very visible, when custom scopes are set to be profiled, that are run many times in a frame. For

example, when enabling profiling for the function that gets the voxel data at a specific location in the chunk, the execution time for the algorithm will increase substantially, since the function is executed 46656 times for building one chunk. This can be seen on Figure 25, that shows that the average execution time for the mesh building function has increased to 6.484ms.

	Inc Time (MS	Inc Time (%)	Exc Time (MS	Exc Time (%)	Calls
AdvDensityTerrainGenerator::GenerateChunk	6.613 ms	48.9 %	0.000 ms	0.0 %	1.5
◢ VoxelChunk::BuildMesh	6.484 ms	48.0 %	8834.724 ms	38.7 %	1.0
▷ VoxelChunk::GetVoxelDataAt	3.664 ms	56.5 %	10965.644 ms	84.9 %	46656.0
Self	2.506 ms	38.7 %	0.000 ms	0.0 %	1.0
▷ VoxelChunk::BuildMesh::AfterCalcIndex	0.225 ms	3.5 %	497.630 ms	62.8 %	93.8

Figure 25. Average execution time of the mesh building function (Inc Time column), with profiling the voxel data lookup function turned on.

4.1.1 Cube Index Calculation Optimization

As is apparent from the performance testing, the operations for determining the cube index are taking up a large part of the execution time for building the mesh. This means that it would be beneficial to prioritize optimization efforts on this part of the code.

The function for calculating the cube index currently performs a voxel data lookup and comparison for every cube corner. This is not entirely necessary, since the algorithm is often calculating the index of a cube that is adjacent to one that was just passed. When iterating through the cubes of a chunk, the innermost loop of the three coordinates is increasing on the z-axis, meaning it is iterating through columns.

Going from a cube to the cube on top of it, the four shared corners of the cube index can easily be reused, since the indices 4 to 7 of the bottom cube, map to the indices 0 to 3 of the top cube respectively as seen on Figure 26. This means, that if the algorithm has already calculated the cube index of the bottom cube, half of the cube index of the top cube can be quickly gained, by just bitshifting the old index to the right by four bits. After this, only the remaining bits 4 to 7 need to be calculated for the top cube, saving multiple voxel data lookup and comparison operations.

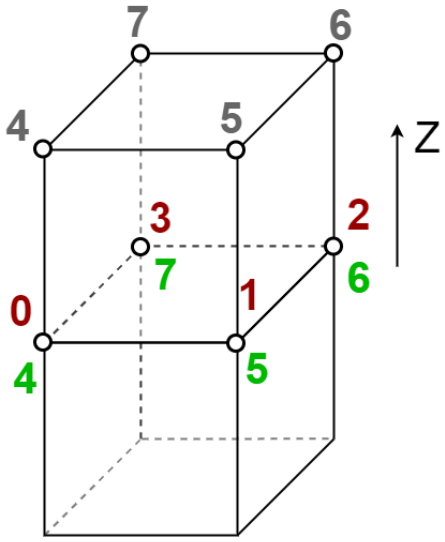

Figure 26. The shared corners of two cubes on top of each other with their indices (red for top cube and green for the bottom cube). The bits 4 to 7 of the top cube that still need to be calculated, are marked in gray.

Profiling the performance of mesh building with this optimization applied, immediately shows significant improvements in execution time. As seen from the graph of Figure 27, which compares the execution time of the function with and without the optimization applied, the time spent on building the mesh has now decreased by a significant amount. The mean execution time for the mesh building function through the test has dropped to 0.808ms. This is a 0.489ms decrease from the results of the unoptimized performance tests from the last section.

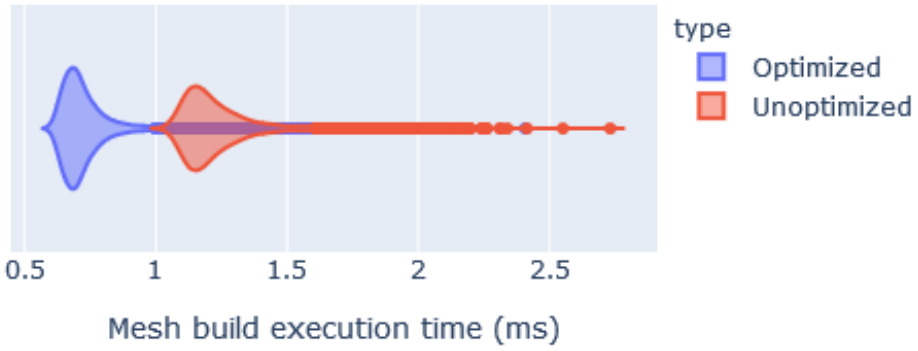

Figure 27. Comparison of the execution time of building the mesh of one chunk with (blue) and without (red) the optimization.

4.2 User Testing

To see what potential users think of the terrain generated, a demo was put together. This demo allowed the user to generate a terrain of a variable size with all three of the terrain generators. It provided default parameters for all three generators, while allowing changing most of the parameters by the user, so that they could experiment with creating different configurations of their own choosing.

A questionnaire, which can be found in Appendix III, was created that provided the demo application and instructions for using it. It then asked the user to generate a terrain with default parameters for each of the generators and score them based on three aspects:

1. How realistic the terrain looked to them
2. If it provided enough variety, or started repeating common features.
3. How interesting the terrain seemed to them

All the scoring questions allowed the user to vote on a scale of 0 to 5 for each generator, where higher numbers were more positive answers. The results of these

questions are shown in Figures 28, 29 and 30. The advanced density terrain generator was scored highest in all three categories. The realism of the density terrain generator was scored noticably lower compared to the other two generators, while the heightmap terrain generator got a low score for variety. When it comes to interest, both the density and heightmap terrain generators were scored quite evenly low.

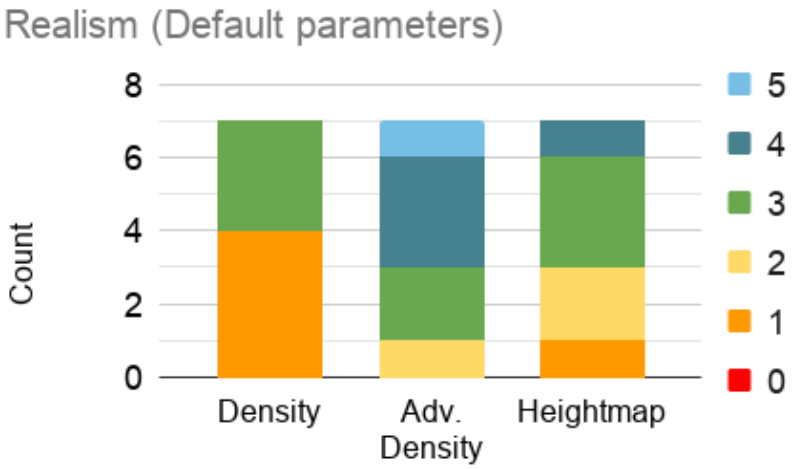

Figure 28. Scores for the realism of the terrain generators.

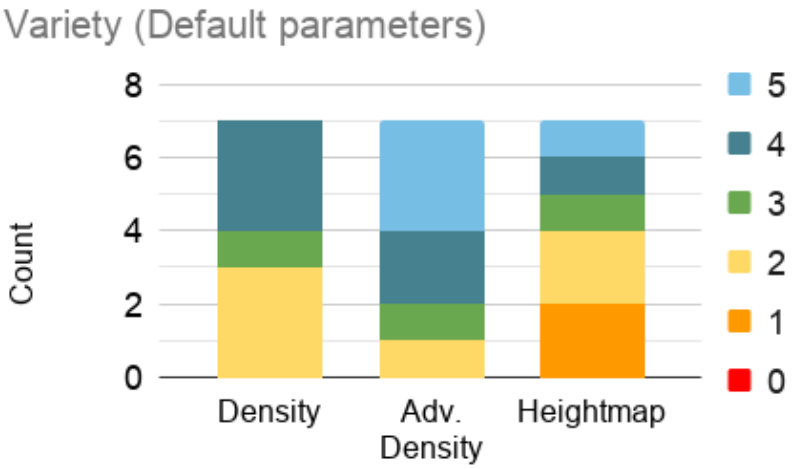

Figure 29. Scores for the variety of the terrain generators.

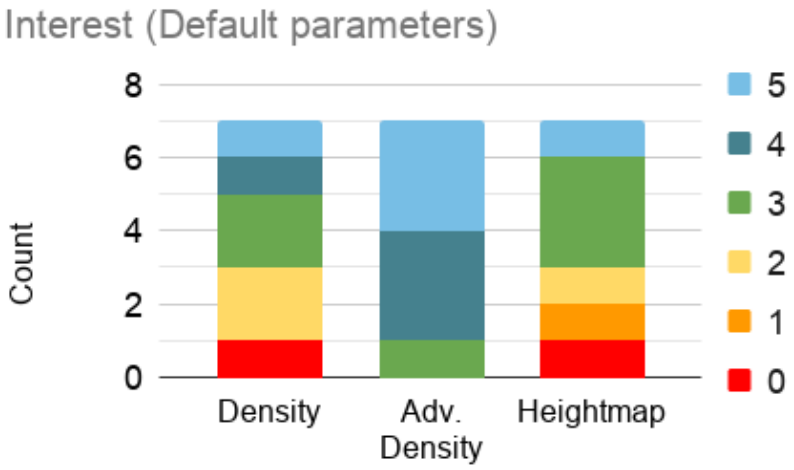

Figure 30. Scores for how interesting the terrain generators seemed.

After experiencing the generators with the default parameters, the users were asked to take each generator and attempt adjusting the parameters to create an interesting terrain, that they would want to explore in a game. After this they were asked to score each generator on a scale of 0 to 5, based on how satisfied they were with the results that they were able to get.

The results of this scoring are shown on Figure 31. The Advanced density terrain generator got high scores here as well, while the other two generators were relatively low, coming to the exact same average score of 2.6. These results suggest that if a game is to be made with this terrain system, then the advanced density generator would be the preferred one to iterate upon in the future.

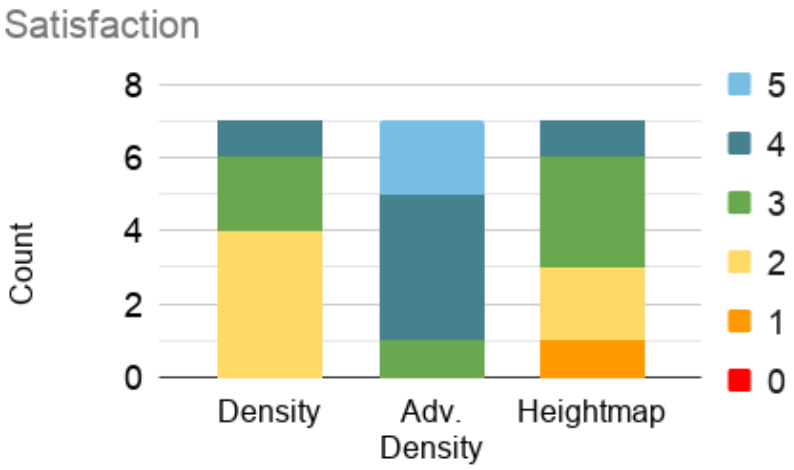

Figure 31. Scores for how satisfied the users were with the results of the generators, after attempting to create a terrain that they would like to explore.

When the users were asked whether they have additional opinions on the system or encountered any problems, multiple of them mentioned sometimes noticing floating pieces of terrain, not connected to any ground. This is a known potential issue with the current system that is not solved in the scope of this thesis. No other technical issues were mentioned.

The demo also recorded the execution time for mesh building and the users were asked to mark down the average mesh build time at the end of their testing. The results of this, along with the hardware configuration of the users, can be seen on Figure 32. The demo was built before the optimizations done in Section 4.1.1. Therefore, these results are more comparable to the observations made in Section 4.1.

Since this performance testing wasn't fully automated in a controlled environment and involved building different terrains for each user, the results cannot be wholly depended upon, but some presumptions can still be made. The average build time before the optimizations does seem to stay around 0.9 to 1.5 milliseconds, going slightly above 2ms on an older CPU. This suggests, that the performance stays more or less in line with the results gained during more in-depth performance testing. The system doesn't seem to slow down enough on different hardware, to become a considerable problem for keeping a steady framerate, while dynamically updating the terrain during gameplay.

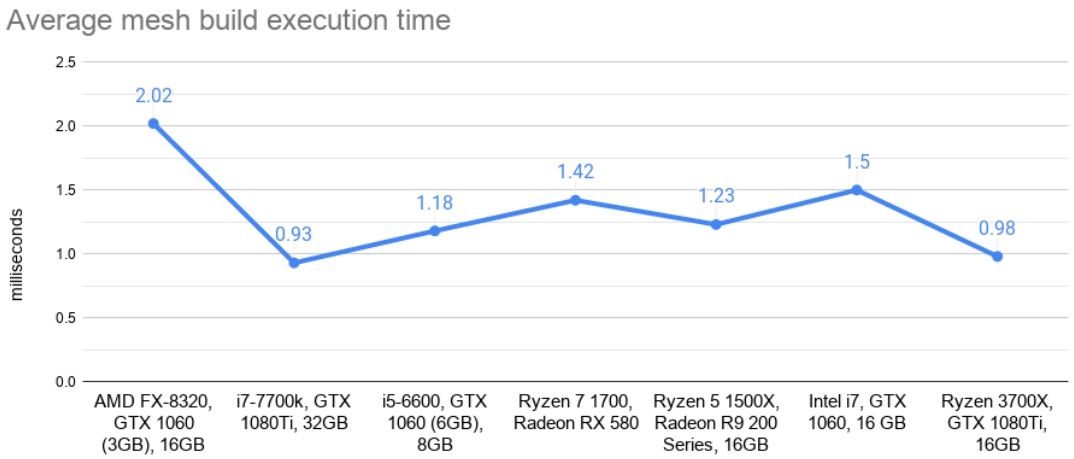

Figure 32. Average execution time for mesh building, noted down by all participants from the demo. Hardware configurations are in the form of CPU, GPU, RAM.

5 Conclusion

As part of this thesis, a dynamic voxel terrain system was built on top of Unreal Engine 4, for use in an in-development game Dwarf Block. At first, voxel visualization techniques were researched. Then, the already existing voxel terrain solutions for Unreal Engine 4, Voxel Farm and Voxel Plugin, were analysed and evaluated for use in the game.

Since no polished solution, that would have been suitable for the game, was found, a custom implementation was built, utilizing the marching cubes algorithm for extracting a triangle mesh out of voxel data, which could then be rendered by the engine. For texturing this procedural mesh, the triplanar projection method was used. The performance of the mesh building was tested and the most performance-heavy part was optimized. From the results of the performance profiling, it was deemed that the system could be used in a real game. This is because it only took a small fraction, of the 16.6ms frame time limit required to stay above 60 frames per second, to build a mesh.

For producing the voxel data used for extracting the mesh, implementations were created for three different terrain generators: density, advanced density and heightmap. For user testing these implementations, a demo application and a questionnaire were created. These were used to compare the different generators based on player feedback. From the results, it was gathered, that the advanced density terrain generator was the most liked of the three. Therefore, it should be iterated upon in the future, to build levels matching the specific theme of the game.

During the testing, no users reported technical problems with the demo. A few users pointed out that floating rocks were generated. In the future, algorithms for preventing this could be implemented along with support for different levels of depth for the mesh and a more efficient structure for holding the voxel data.

References

[Blo] Cory Bloyd. http://paulbourke.net/geometry/polygonise/marchingsource.cpp. (07.05.2020).

[Gei07] Ryan Geiss. Generating complex procedural terrains using the gpu. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 1. Addison-Wesley Professional, 2007.

[Gre09] Gerrit Greeff. Interactive voxel terrain design using procedural techniques. Master's thesis, Stellenbosch: University of Stellenbosch, 2009.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[Owe] Brent Owens. Use tri-planar texture mapping for better terrain. https://gamedevelopment.tutsplus.com/articles/use-tri-planar-texture-mapping-for-better-terrain--gamedev-13821. (08.05.2020).

Appendix

I. Glossary

1. **Frame time** - Time spent by the game between rendering two frames.
2. **Flat shading** - Technique, where each triangle of a mesh has its own vertices, which have normal vectors pointing in the same direction. This gives the triangles a flat look and hard transitions between triangles.
3. **Fragment** - Segment of a mesh's triangle, interpolated from its vertices. Typically each fragment corresponds to one pixel on the target render frame.
4. **Interpolation** - Process of estimating new data points based on already defined data points.
5. **Noise function** - Function which based on an input of one or more coordinates, outputs a pseudorandom value.
6. **Smooth shading** - Technique, where adjacent triangles of a mesh share vertices. Opposed to flat shading, each vertex here has its own normal vector value. This gives the mesh smooth transitions between the triangles.
7. **Triangle Mesh** - Collection of vertices, connected to form triangles, used to render a shape.
8. **Vertex** - Point in a vector space. Typically used to define corners of polygons of a mesh. Contains attributes that define properties of the mesh at the given point.
9. **Voxel** - Element, containing a set of attributes, that represents a volume in a three dimensional vector space.

II. Demo User Guide

The demo application can be found within the attachment zip file, in the folder "Demo". The application is built for running with 64-bit Windows. For running the application, execute the file "Dwarf Block - Terrain.exe" in the folder. The controls are as follows:

Table 1. Controls for the demo application.

Key	**Function**
W,A,S,D	Movement.
E,Q (or Ctrl, Space)	Move up/down along the world's vertical axis.
Scroll wheel up/down	Increase/decrease movement speed.
R or Escape	Change between movement mode and UI mode.

The player starts in movement mode and can use the mouse to look around. By pressing R or Escape, UI mode can be toggled on, which allows interacting with the UI elements using the mouse and keyboard.

Parameters of the terrain generator can be set in the panel in the top-right corner of the screen. Below this panel is a dropdown, where the terrain generator type can be changed. For the changes to take effect, the "Generate" button must be pressed, to start generating a terrain with new parameters. Application can be exited with the button in the botton-right corner.

This demo application includes the performance optimization of Section 4.1.1.

III. User Testing Questionnaire

The questionnaire can be found in the attachment zip file, as a pdf file named "questionnaire.pdf".

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Märt Mäemees**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Dwarf Block Game Development - Dynamic Environment,

   supervised by Jaanus Jaggo.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Märt Mäemees
08/05/2020