Game Engine Architecture Blocky Terrain Generation

Tom Smith
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
Thomas19.Smith@live.uwe.ac.uk

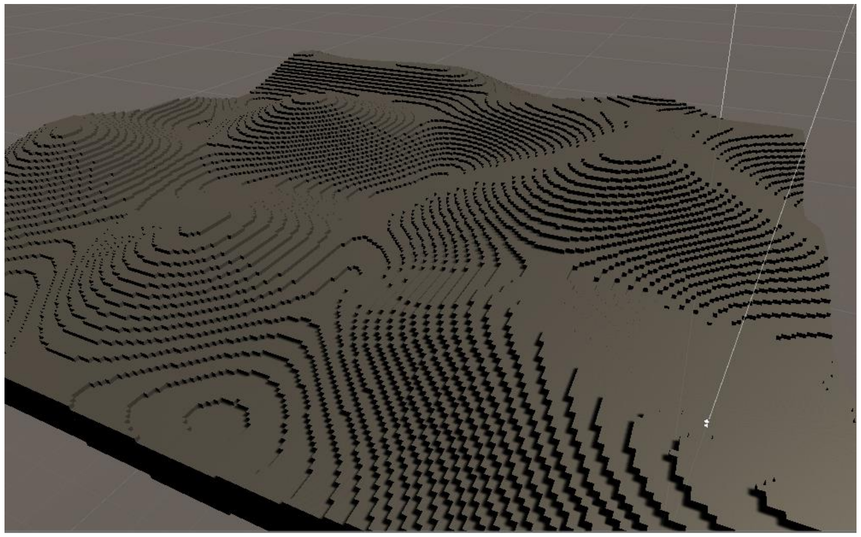


Figure 1. Terrain generated in the Unity Engine

Abstract

This document describes how a system to generate random terrain a voxelated or blocky style was implemented in the Unreal Engine 4 and the Unity Game Engine. It details how the overall system works, as well as explaining which features of the engines were used and to what effect. Ideas formed from research are explained, as is the reason for their inclusion within the implementation of the system, (for example noise algorithms and mesh generation). To conclude, an evaluation of the systems implementation in both engines is featured to outline their overall effectiveness.

Author Keywords

Terrain Generation; Procedural; Unreal Engine; Unity Engine; Terrain in Games; Noise;

Introduction

The procedural generation of terrain is something utilized in games to create large worlds for the player to travers without having to model the world themselves and store the model in each version of the game. It is most notably used in the massively popular survival game *Minecraft* (2011). This game featured a blocky art style and rendered terrain, much like the systems implemented, as such – meaning the ground was formed by cubes, giving the entire landscape a voxelated aesthetic. By generating landscape procedurally and randomly, there becomes available the option for a seemingly infinite 3D game world for the player to explore. To generate the terrain, both systems employ a noise algorithm to generate random rolling hills and troughs in the terrain that link seamlessly with the surrounding landscape. Utilizing random seeding for the algorithms, the terrain forms in a seemingly random manner each time it is executed.

Background & Research

Implementations in other games:

There are many games that utilize terrain generation, but most notable is in Swedish developer Mojang's *Minecraft* (2011). In Minecraft the terrain is generated in chunks of 16x16 blocks, this is to optimize performance and allow the player to set a 'render distance', that dictates how many chunks are rendered at a time for the player and allows lower end systems to still run the game. An easy way to think of the way Minecraft organizes its map is to imagine a 3D grid in which blocks are placed, all X and Z positions must have a block, but in order to generate altitude changes within the terrain, the Y coordinate for each corresponding (X,Z) coordinate can just be increased or decreased to generate an inclining or declining gradient.

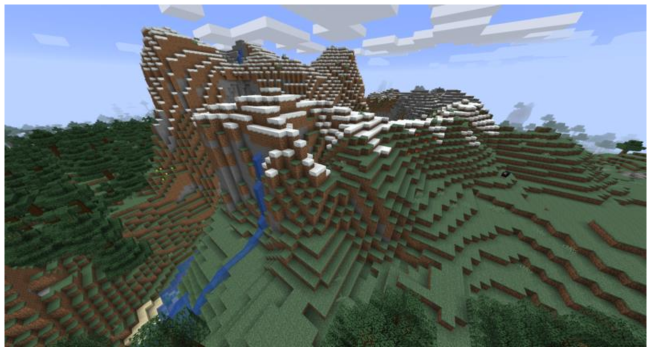


Figure 2. Terrain generated in Minecraft

It is clear that this implementation within Minecraft is exactly what the systems aim to replicate, it was just a case of deciding how to calculate and apply the Y displacement needed to form gradients within the landscape. That is where noise algorithms come in.

Perlin/Simplex Noise:

Perlin noise is an algorithm created by Ken Perlin in 1983, used to generate texture using mathematics, these are referred to as procedural textures (see figure 3). A procedural texture is what is known as 2-dimensional noise, and in order to generate landscapes, and extra dimension is added, making 3-dimensional noise. This is achieved by considering the procedural texture formed by the Perlin algorithm as a height map, where the lighter areas of the texture represent higher altitudes and darker areas represent lower altitudes (see figure 4).

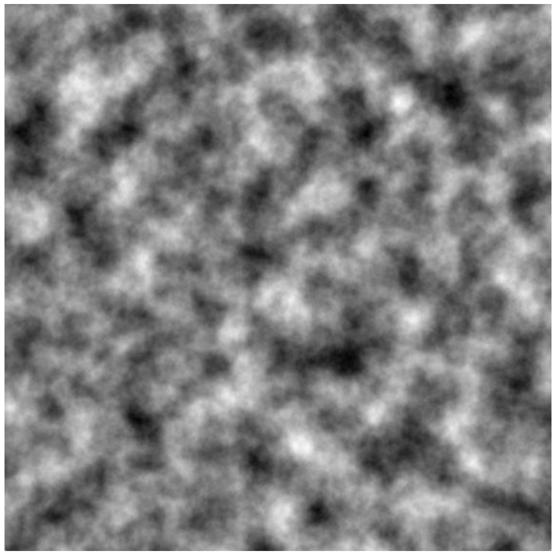


Figure 3. Procedural texture generated with Perlin noise

Figure 4. A 3D landscape generated using Perlin noise

The Perlin noise function returns a continuous waveform of values between 0 and 1, and by decreasing the increments between the values you send to the function the smoother the outputted results. Hence, if you consider the outputted results to be a wave, you can increase or decrease the frequency to produce more jagged and erratic terrain, or smoother terrain respectively. Another thing to consider when generating terrain is the maximum Y displacement the blocks can have, essentially the highest peak a block can be, this would be the amplitude of the wave and is as simple as multiplying the output of the Perlin function by the desired max height.

While Unity has a built-in function for calculating Perlin noise, Unreal does not. This complicated the implementation in the Unreal Engine, however accessible online is a plugin for the engine to implement simplex noise, created by GitHub user *devdad*. This adds functions to generate simplex noise in Unreal. Simplex noise is a newer alternative to Perlin noise also created by Ken Perlin in 2001, it achieves mostly the same results with some optimizations (optimizations that were not necessary for the scope of this project, hence why Perlin is used in Unity).

With this as the base for generating the terrain within the engines, it was clear that optimizations were needed to allow the system to run smoothly – especially if implemented in a wider game environment. A solution to this problem is used in Minecraft to allow for such vast amounts of terrain to be generated and rendered in game: mesh generation.

Mesh Generation:
Instead of having each block in the terrain be a separate entity with all 6 sides rendered, it is far more efficient to simply map a mesh over the exposed surface of the terrain, and render only the visible sides of the cube as one continuous shape – almost like welding the cubes together and discarding the sides the player cannot see. The engines chosen to do this have built in functionality for mesh generation combining meshes.

Implementations

When implementing the terrain generation in both engines the general idea was to generate the terrain by placing instances of a block in a NxN grid and change each instance's altitude based on the output of a Perlin/Simplex noise function. Then a mesh is created with the 'mold' formed by the blocks which can then be deleted for optimization. This should easily allow the production of square regions of terrain repeatedly.

Unity Game Engine Implementation
Assets Used:

- **FPSController (Obj)-** player character used to move around the terrain generated, obtained via Unity's default assets.
- **Terrain(Obj) –** An empty object that generates a 50x50 chunk of terrain.
- **Main menu(Obj) –** The canvas on which the title and buttons of the main menu are displayed.
- **Game UI (Obj)–** The canvas on which the game UI is shown, simply a small bit of text with instructions for use.
- **EventHandler (Obj)–** An empty object that manages the functions called by menu buttons.
- **Main Camera (Obj)–** The camera used in the menu, once past the menu the player character has a camera attached.
- **Light (Obj)–** Directional light to illuminate the terrain.
- **Grass (Mat) –** One of three available materials for the terrain.
- **Sand (Mat) –** One of three available materials for the terrain.
- **Stone (Mat) –** One of three available materials for the terrain.

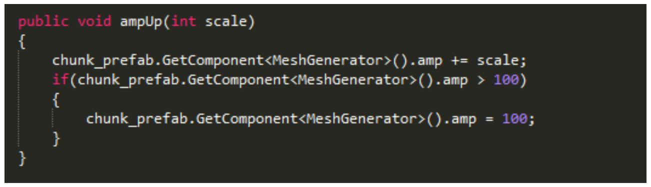
Features:

- A menu screen allowing the choice of the material of the terrain, the amplitude, the frequency, and the size of the chunk of terrain generated.

- A UI prompting the player to click X returning them to the main menu allowing regeneration of terrain any number of times.
- A first person character the player uses to traverse the terrain.
- Random seeding of the noise function meaning generated terrain is always different.
- Frequency limited to between 2 and 400 to allow for suitable (fully traversable) terrain to be generated.
- Amplitude limited to between 1 and 100 to allow for suitable (fully traversable) terrain to be generated.
- 3 size options for terrain generation, 150x150, 200x200, and 250x250.
- 3 different materials for the terrain, grass for fields, sand for deserts, and stone for mountains.

Code Breakdown:

Upon starting the system, the user is presented with a menu screen, this is where variables for the terrain generation are set. Buttons in the menu call functions when clicked that increase/decrease the amount of chunks generated, or the frequency/amplitude passed into the Perlin function. The values for amplitude and frequency are restricted to maximum and minimum values. There is also a button to cycle through the material used on the cubes forming the terrain, this works the same way as the size buttons in that it just iterates through an array of available options.



```
public void ampUp(int scale)
{
    chunk_prefab.GetComponent<MeshGenerator>().amp += scale;
    if(chunk_prefab.GetComponent<MeshGenerator>().amp > 100)
    {
        chunk_prefab.GetComponent<MeshGenerator>().amp = 100;
    }
}
```

Figure 5. Example of how values are limited in the case of button presses.

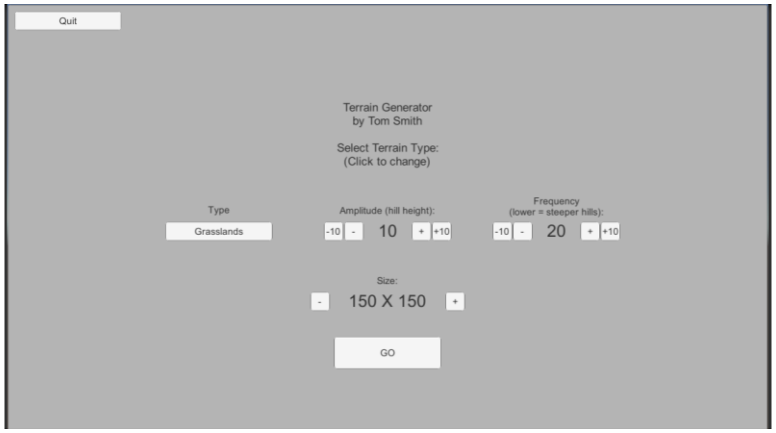
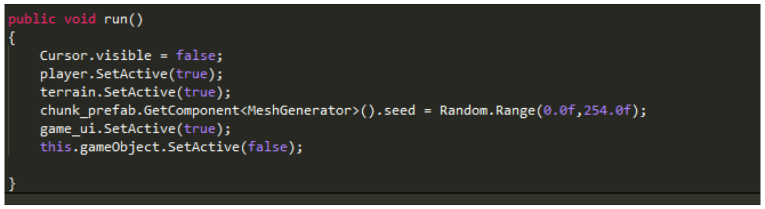


Figure 6. The menu shown as the start of the system

Upon clicking ‘GO’, the first instance of ‘Terrain’ is activated, alongside the player and the UI. The menu screen is then deactivated.

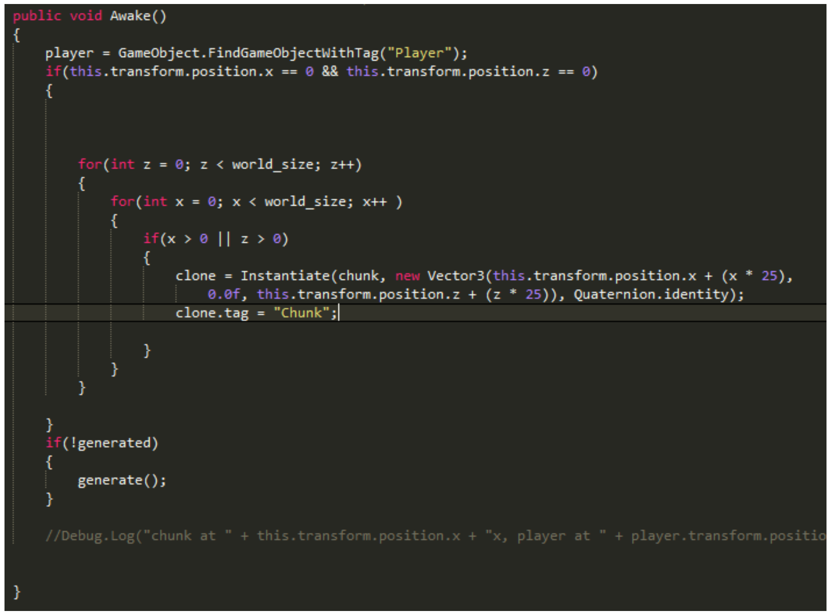


```
public void run()
{
    Cursor.visible = false;
    player.SetActive(true);
    terrain.SetActive(true);
    chunk_prefab.GetComponent<MeshGenerator>().seed = Random.Range(0.0f,254.0f);
    game_ui.SetActive(true);
    this.gameObject.SetActive(false);

}
```

Figure 7. The function called upon clicking go, enabling and disabling the relevant game objects.

In the 'Terrain' game object's script, there exists an Awake() function, this is a function built into Unity that is called as soon as the game object to which the script is attached is enabled, or instantiated. This is the first thing is does when the GO button is pressed and it becomes active, so this is where it takes the world size set in the menu, and instantiates the right amount of clones of itself to create a chunk of terrain the specified size. Each 'Terrain' object produces a 50x50 chunk, so depending on the world size this awake function instantiates between 8 and 24 clones of itself in a grid.

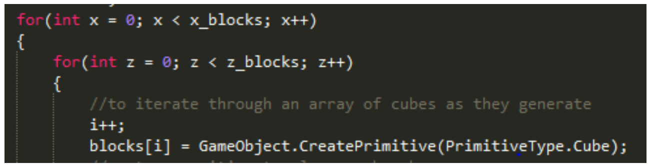
This section of the awake is enclosed in an if statement that means it is only called in the very first instance of 'Terrain', otherwise each instance will then try to instantiate more clones, causing them to generate infinitely.



```
public void Awake()
{
    player = GameObject.FindGameObjectWithTag("Player");
    if(this.transform.position.x == 0 && this.transform.position.z == 0)
    {



        for(int z = 0; z < world_size; z++)
        {
            for(int x = 0; x < world_size; x++ )
            {
                if(x > 0 || z > 0)
                {
                    clone = Instantiate(chunk, new Vector3(this.transform.position.x + (x * 25),
                        0.0f, this.transform.position.z + (z * 25)), Quaternion.identity);
                    clone.tag = "Chunk";

                }
            }
        }

    }
    if(!generated)
    {
        generate();
    }

    //Debug.Log("chunk at " + this.transform.position.x + "x, player at " + player.transform.positio


}
```

Figure 8. Awake function within Terrain script.

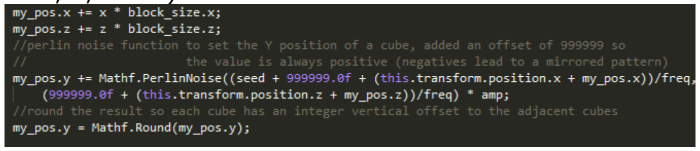
This then calls the generate() function, which is where all the blocks are generated to form the terrain. The first step of this process is to set the scale of the cubes that are used to generate the land scape. Originally, this was set using the vector(1.0,1.0,1.0), meaning it was a uniform cube of width, height, and depth of 1.0, however this led to gaps in terrain forming if there was a steep incline where one block was 2 blocks higher than the one next to it. To fix this issue the blocks are generated with a height of 10 so they go far below the reasonable expected change in altitude of the adjacent blocks. The next step is to generate an array of game objects of size 250 (note this is not hard coded, the value of x_blocks and z_blocks default at 50, but can be changed to reduce the size of each Terrain objects chunk, however 50x50 is the maximum). Then 2 for loops are used to iterate through all 250 empty game objects in the array, create blocks and place them in a 50x50 grid in the game world.



```
for(int x = 0; x < x_blocks; x++)
{
    for(int z = 0; z < z_blocks; z++)
    {
        //to iterate through an array of cubes as they generate
        i++;
        blocks[i] = GameObject.CreatePrimitive(PrimitiveType.Cube);
```

Figure 9. The for loop used, with blocks being created and placed within the established array.

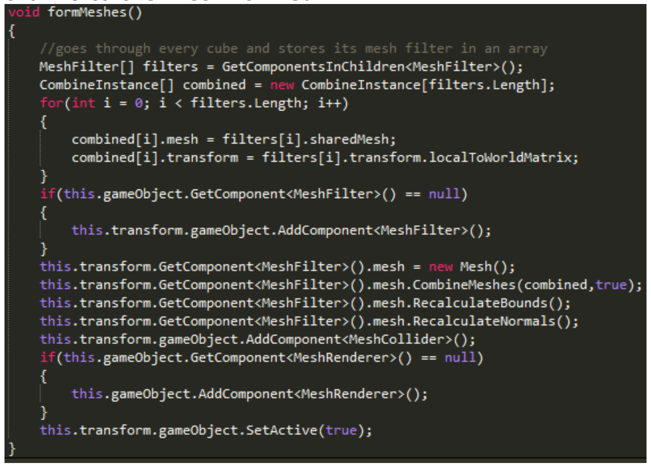
The y position of each block is incremented by the result of a built in PerlinNoise function in Unity. The x and z coordinates, the frequency and amplitude (as set in menu), and the seed that is randomly generated upon clicking GO, are all passed into the function, and the result is the Y displacement for that block. The result is then rounded to the nearest integer to ensure the block conforms to the uniform distribution rules (1 block cannot be 0.3 blocks higher than another, it must be 0, 1, 2 etc.).



```
my_pos.x += x * block_size.x;
my_pos.z += z * block_size.z;
//perlin noise function to set the Y position of a cube, added an offset of 999999 so
//                        the value is always positive (negatives lead to a mirrored pattern)
my_pos.y += Mathf.PerlinNoise((seed + 999999.0f + (this.transform.position.x + my_pos.x))/freq,
    (999999.0f + (this.transform.position.z + my_pos.z))/freq) * amp;
//round the result so each cube has an integer vertical offset to the adjacent cubes
my_pos.y = Mathf.Round(my_pos.y);
```

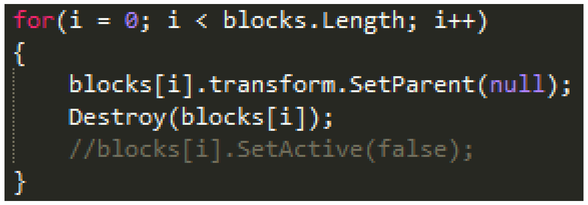
Figure 10. The position of the current block in the array being set, showing how the PerlinNoise function is used.

Once each of the blocks have been placed to form the terrain, the next function called is formMeshes(). This uses components built in to Unity's meshes to combine the meshes of the blocks into one mesh, set new boundaries and establish a collider for the newly generated mesh. Then all the blocks are destroyed to increase performance, as they are now unnecessary thanks to the mesh formed.



```
void formMeshes()
{
    //goes through every cube and stores its mesh filter in an array
    MeshFilter[] filters = GetComponentsInChildren<MeshFilter>();
    CombineInstance[] combined = new CombineInstance[filters.Length];
    for(int i = 0; i < filters.Length; i++)
    {
        combined[i].mesh = filters[i].sharedMesh;
        combined[i].transform = filters[i].transform.localToWorldMatrix;
    }
    if(this.gameObject.GetComponent<MeshFilter>() == null)
    {
        this.transform.gameObject.AddComponent<MeshFilter>();
    }
    this.transform.GetComponent<MeshFilter>().mesh = new Mesh();
    this.transform.GetComponent<MeshFilter>().mesh.CombineMeshes(combined,true);
    this.transform.GetComponent<MeshFilter>().mesh.RecalculateBounds();
    this.transform.GetComponent<MeshFilter>().mesh.RecalculateNormals();
    this.transform.gameObject.AddComponent<MeshCollider>();
    if(this.gameObject.GetComponent<MeshRenderer>() == null)
    {
        this.gameObject.AddComponent<MeshRenderer>();
    }
    this.transform.gameObject.SetActive(true);
}
```

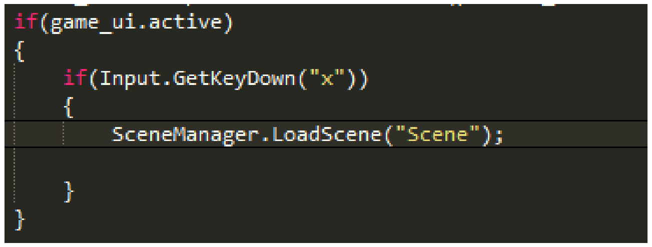
Figure 11. The formMeshes function.



```
for(i = 0; i < blocks.Length; i++)
{
    blocks[i].transform.SetParent(null);
    Destroy(blocks[i]);
    //blocks[i].SetActive(false);
}
```

Figure 12. Upon generating the mesh, the obsolete blocks are iterated through and destroyed.

Finally, the event manager is constantly checking for the user to input 'X', this triggers the scene to reset, returning to the menu and reinitializing all objects and variables. This allows for the system to run and rerun any number of times.



```
if(game_ui.active)
{
    if(Input.GetKeyDown("x"))
    {
        SceneManager.LoadScene("Scene");

    }
}
```

Figure 13. The event manager checks, only when in game as the game UI must be active, every frame for an X key press at which point the scene is reloaded and the system is reset.

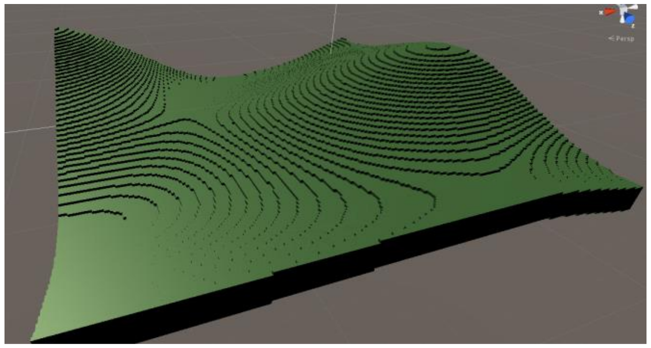


Figure 14. Terrain generated in Unity, with grassland type, 60 amplitude and 130 frequency.

Unreal Engine 4 Implementation:
Unreal was a previously unused development environment, and so the system implemented in this engine is far less complete than the Unity implementation. This alongside the lack of a built in Perlin noise function meant that the implementation differed significantly from the first.

Assets used:

- **ThirdPersonExampleMap –** A default base project for a third person character.
- **Light source –** Light to illuminate the generated terrain.
- **ThirdPersonCharacter –** The character controlled by the user that traverses the generated terrain.
- **Voxel –** The object used to generate the terrain.
- **M_Ground_Grass -** Material applied to the terrain generated.

Features:

- Third Person character to navigate the generated terrain.
- Repeatable, random terrain generation.
- Ability to regenerate the level at the press of a key.
- UI to instruct user how to reset level.

Code Breakdown:

Upon the system starting, the player character is spawned in the air above the terrain, this is to account for diverse altitudes of the block below the players spawn point. The game object voxel has an event check called 'beginplay' that executes the terrain generation function as soon as the level is built.

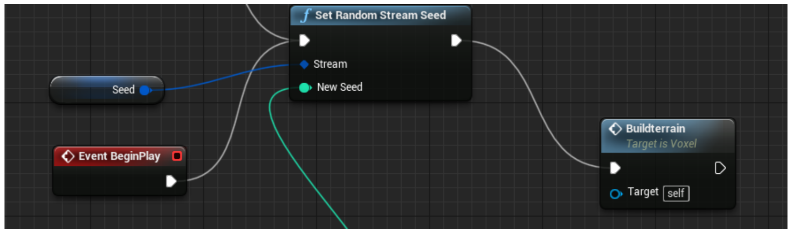


Figure 15. Shown here is the immediate generation of a random seed for the generation and then the function to generate being called when play begins.

The seed is used for all random generation within the buildterrain function, it is set using the system clock to ensure the seed is almost always different each time it is set. The buildterrain function initially produces 3 random values, two are placed into the variables 'voxel Density' and 'Factor', these are used in the generation of simplex noise, (similar to amplitude and frequency used in the Perlin function in Unity), and the third is placed into 'Ran Displacement'. This is used like the seed was used in the Perlin function, and just acts as an extra level of randomness in the generation by adding it to the coordinates values passed into the Simplex function. It then sets the size of the chunk to 150 blocks, this will actually be 150x150 as a chunk is always a square. This is where two for loops are used to iterate through each position in the 150x150 grid and sets the corresponding coordinates. Within the loops, it then generates the Simplex Noise for each position in the grid, and calls the function 'Add Noise'.

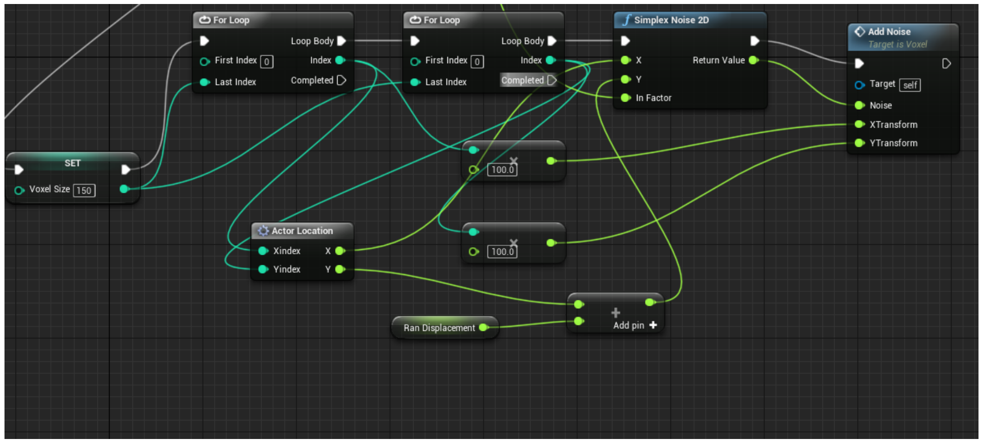


Figure 16. The for loops shown that iterate through 150, note that Actor Location is used to output the x and y coordinates of the actor, ignoring the z which is the altitude. The result of this would be the add noise function being called 22500 times, to place all the blocks in the position defined by the loop.

The Add Noise function simply takes the XTransform, ZTransform, and Noise passed into it and forms a vector from them, passing it into a Add Instance function that creates an instance of the mesh cube attached to the voxel actor. To tackle the issue of gaps appearing in terrain, it actually places 2 blocks on top of each other at each location.

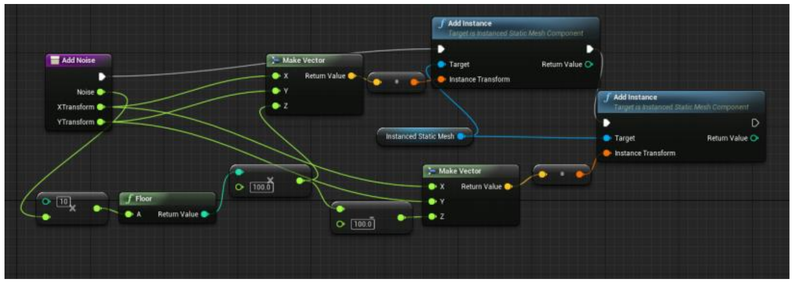


Figure 17. The Add Noise function.

The Voxel also has an event check for pressing the X key, this resets the level to allow for regenerating the level.

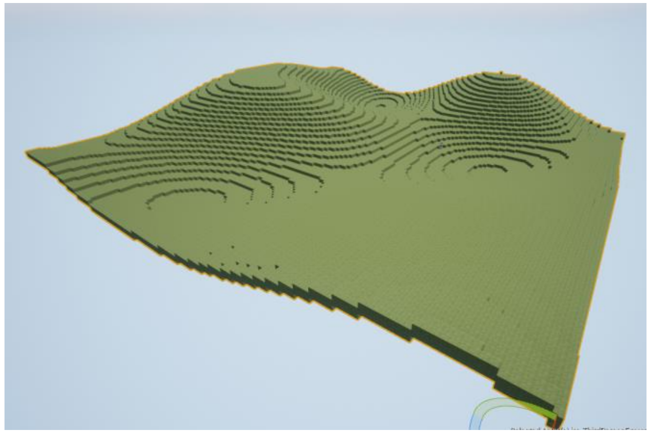


Figure 18. Landscape generated in Unreal Engine.

Evaluation

The general plan for developing the effect was consistent between both engines, however the Unreal implementation was severely lacking in terms of customization, as well as general quality. There is no mesh generation present in the Unreal implementation, and performance suffers accordingly, the terrain also takes a long time to generate, with the user clicking X to reset the level often resulting in the program hanging for several seconds before anything occurs.

The biggest issue shared by both engines was the repetitive and predictable 'random' generation. Seeding random generation in Unreal was especially problematic, as the effect in-editor would often function differently to the packaged executable, as if the editor was more forgiving. Terrain would always generate identically when packaged as an executable, this is due to the set random seed function not working beyond the first call in the voxel blueprints, however in editor this was not an issue. The solution was to add a C++ script simply with the line srand(time_t(NULL)), this now seeds the random using the system clock every time the level is built, allowing random terrain to be generated in the executable and in editor. A similar issue occurred in Unity when the setting of the random seed for the Perlin function was in the wrong place, it led to all chunks being a perfect clone of each other, or not lining up at all. The solution to this was simply to set the seed for the function when the GO button was clicked.

There are many ways in which the implementation can be improved, for example adding a chunk loading system where in the user can move around infinitely and chunks will be loaded and unloaded around them as they do so. Also there can be more customization options, particularly in the Unreal version, as well as a more sophisticated menu in both solutions. Performance needs to be addressed in the Unreal implementation as while it functions for its current purpose it would not be usable elsewhere due to its resource demands and inefficient solution.

I think the strongest aspect of the Unity solution is its customization options, providing a more in depth generation and also opening the door to potential additional systems such as adding biomes and changing terrain characteristics with location. The Unreal solution's biggest strength is its simplicity, while very basic it is easy to use and works well as an introduction to terrain generation, and is a clear demonstration of Simplex noise.

Both systems could easily be used in a wider game environment for generating the levels, due to very few of the variables being hard coded, it is easy to edit the characteristics and with all the code and process being contained within one placeable and instantiable object/actor, the reproduction of chunks is quick and easy.

After implementing the system in both engines, it is clear Unity produced the more sophisticated solution with far more features and depth, however this is more indicative of the prior experience in the engines than the engines themselves. However, due to the built in Perlin noise function, Unity is the better environment for implementing this effect. Overall, both engines implemented the effect successfully, both meet the initial goal set, the Unity implementation was just more in-depth.

References

Mojang (2011) *Minecraft* [Video game]. Available from: https://www.minecraft.net/en-us/store/minecraft-java-edition/

Devdad (2020) *SimplexNoise* [computer program]. Available from: https://github.com/devdad/SimplexNoise

Holistic3d (May 27, 2018) *The Theory of Noise: An Overview of Perlin Noise* [video]. Available from: https://www.youtube.com/watch?v=H6FhG9VKhJg&ab_channel=Holistic3d

Scratchapixel (n . d) *Perlin Noise: Part 2* [Article]. Available from: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2

Unity Technologies (n . d) *Unity Scripting API: Mathf.PerlinNoise* [Article]. Available from: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

Shojib Tutorials (Aug 18, 2018) *Unreal Engine 4 – Minecraft-like Landscape Using Noise* [video]. Available from: https://www.youtube.com/watch?v=1VLVNYR-lRc&ab_channel=ShojibTutorials