Perlin Noise Map Generation

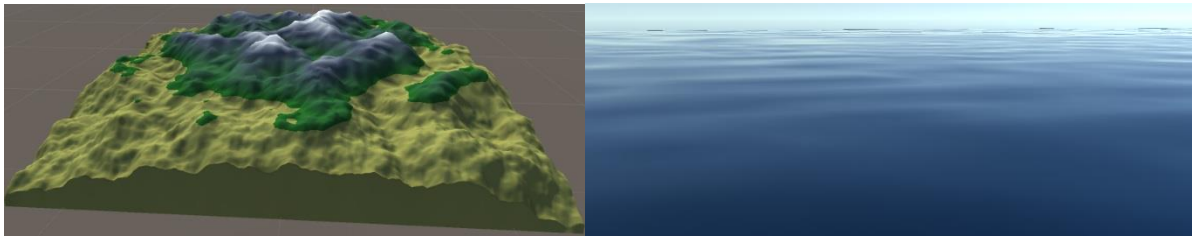
For the Assignment 2 project, I decided to focus my efforts on procedurally generated content that has consistency upon generation. This wold primarily rely on computer based pseudorandom number generation in order to generate Perlin noise maps in which data could be extracted. The applications of using procedurally generated content in games allows for a longer effective shelf life of a product, wherein each game may act as a relatively new experience while having consistency within the gameplay loop and mechanics. However, the usual drawback for using procedural content is that there is a lower overall quality for level design, wherein without specific precautions, it is possible to create unplayable maps or have all terrain elements be clustered in one specific area creating a blank space in design that would be bothersome to players.

For this assignment I believed that programming using C# in the Unity engine was ideal due to the pre-existing Perlin noise function under the Mathf class while the Unreal engine would require a plugin or blueprint system to generate the noise prior to coding the project using UE C++, this coupled with the inspector component for variables allowed runtime modifications to the Perlin noise generation script to find the best combination of inputs for the terrain.

The main programming technique used in this project is the use of a Perlin noise generator for both the water effects by the islands generated and the island generation in itself. Normally, Perlin noise would generate a random heightmap of realistic looking terrain without any particular features being above any of the others, however, by adding a base level of an inverted elliptic paraboloid*, it is possible to force the generation on top of a dome, rather than a flat surface. Then, through manipulation of constants such as the lacunarity and persistence of the Perlin noise it is possible to ensure that the heightmap would almost always create an island with at least one mountain peak at the maxima. However, the current approach will never produce any form of cavern or cave system without further work, though it is possible for other features such as small islands, bays, and mountain ranges to be prominent.

*Inverted ecliptic paraboloid calculated using a formula of Heightmap Index = $n – x^2 – y^2$ where x and y are references to the point's index in the 2D array and n is a constant representing the maxima of the paraboloid.

(Below: An image of the complete generated terrain component and the Perlin nose water effect)



Following this I have also added a system that reads the generated heightmap and creates a 2d array of the maximum gradients at each point on the map to be stored in the terrain data, as each point of terrain is stored on a 2D heightmap, a script has been added that reads each index in the array, compares all opposing neighbouring points (X index increased by 1 with X index decreased by 1 to calculate a "vertical" reference) and stores the greatest gradient calculated in the 2D array. For the

edges of the heightmap, the code has been set up to prevent index out of bounds errors by referencing only accessible indices and logically calculating how many indices have been used to calculate the gradient at a given point (As all nodes in the graph are the same distance from each other, there is no need to store the data as Vector3 variables, as floats would contain the same effective data with less processing power used in calculation and less memory allocation). This could then be used to generate roads that follow the path of the lowest gradient as well as the shortest route to create a more natural looking environment, this would be possible with the application of an A* pathfinding script that reads steeper gradients as inaccessible points on the 2D array. Coupling this with the accumulative gradients from the traversal, the path created would be the flattest route between any two points.

Beyond this I have also created a modifiable shader that reads the height of each point on the map in order to determine an appropriate colour scheme, blending the snow of the mountain peak into the rock layer, and slowly blending into grass before suddenly becoming the shoreline beach. The shader itself could be used to apply a variety of textures to the terrain, however I preferred the simpler aesthetic of plain colours. The reading of the terrain used in the shader could also be extended to the placement of foliage, such that at low level of grasslands it is possible for forests to spawn, while the more mountainous regions will consist of rock formations.

As shown by the images on the previous page, the terrain will consistently generate in a manner that prevents cut-off at the edges of the terrain while also allowing the possibility of multiple instances of islands to be generated alongside one larger, central island. In doing so it is possible to create randomly generated terrain in a controlled manner, however, the project as it stands does not specifically contain code to generate foliage and details on top of the main island, which would need to reference the height based biomes set up by the generation of terrain (As shown through the shader setup).

For the project I have submitted I would aim to implement this terrain generation into a “Battle Royale” game akin to Player Unknown Battlegrounds and Fortnite Battle Royale, the benefit being that the map may contain the same base elements in each game session by adding buildings to the map upon creation, but the player must consistently adapt to a new map layout each game, preventing the ability to have a set metagame in effect, while for further implementing the terrain generation aspect, I would need to add the already discussed foliage tools as well as implement a system that can alter the terrain in order to place buildings and town areas if this was to be applied to the Battle Royale genre, this could be accomplished by calculating an average height of an area and using a lerp function to smoothen and flatten the terrain before placing objects around the central point.