Procedurally Generating RTS Maps

Yu Cheng – 100885508

Supervisor: Prof. Oliver van Kaick

December 15, 2017
Abstract

Many real-time strategy games have generic looking maps, which can potentially be generated instead of made by a person. In this project I created a script which procedurally generates a mesh usable for real-time strategy games. It done using a heightmap generated from Perlin noise while using methods used in generating mountainous terrain to adjust the result. The terrain is then made to look like something from a game by smoothing out the values to two fixed elevations, and adding strategic features for players to play around. The result of the project is a script that can generate a map using variables which can be manipulated on the Unity engine editor.
I would like to express my gratitude for Professor Oliver van Kaick for giving me the opportunity to complete and supervise this project.
Table of contents

3 – Abstract
4 – Table of contents
5 – List of figures
6 – Motivation
10 – Methodology
25 – Result
28 - References
List of figures

7 - Figure 1 – Top down view of a map in the Starcraft campaign
8 - Figure 2 - A generic two player map in Company of Heroes
12 - Figure 3 – Visualization of the Perlin noise algorithm
12 - Figure 4
12 - Figure 5
13 - Figure 6 – Noisemaps generated with and without octaves
13 - Figure 7
13 - Figure 8 – Visualizations of terrain generated with and without octaves
13 - Figure 9
14 - Figure 10 – Noisemaps generated with different frequency values
14 - Figure 11
15 - Figure 12 – Screenshot of the code block that uses Perlin noise
17 - Figure 13 – A example of an animation curve setting
17 - Figure 14
18 - Figure 15 – High and low ground highlighted in different colors
20 - Figure 16 – Example of a pillar
21 - Figure 17 – Example of a ramp
22 - Figure 18
23 - Figure 19
25 - Figure 20
26 - Figure 21
Motivation

In the genre of real-time strategy games, also known as RTS for short, players compete by building up large bases and armies to destroy the other player’s base. This genre usually has the player looking in a top-down perspective on their army and base, with the setting being some landscape with hills, high ground, cliffs, ramps, or other doodads that obstruct the armies. These features are present to force the player to make strategic decisions on where to build certain things, how to position armies, or where the best place to have their army fight is. Replay-value in RTS games rely heavily on different maps, since the core mechanics are the same in every playthrough. Because of this, the maps must be different to some extent so the gameplay between each map feels different, and the player is able to get more enjoyment out of the same game but with different levels.

There are different types of RTS games, but almost every game tends to use one of two styles of maps. Blizzard and Westwood, two of the most well-known pioneers in the genre, use maps that resemble real life terrain but make heavy use of strategic features, like small ramps that head into a high ground area, or obstructions like trees and rock formations. As shown in figure 1, a map from a mission in the original Starcraft game by Blizzard, there’s generally 3 kinds of terrain one will encounter. One of them cannot be walked upon by grounded units, but the other two are traversable, and have different elevations. Every high ground area in this example has a ramp that allows ground units to access it, but there are cases where a high ground area will have no ramp, but contain a sizable cache of resources on it as reward for a player who can construct a base there there.
Figure 1 – Top down view of a map in the Starcraft campaign

Other games such as Company of Heroes or Warhammer tend to use terrain that are very similar layouts to what you would expect from a real battlefield, and have generally flat terrain riddled with buildings and manmade looking obstacles.
I have been a long time player of Blizzard and Westwood RTS games, and throughout my experience of playing different titles from this style of RTS games, I’ve noticed many of the campaign and single player maps share similar features. Particularly in older RTS titles, I find that stock maps and scenarios don’t focus too much on the terrain aspect of map building, but rather on giving the player a limited amount of units or technology they can use for a particular mission. Due to these patterns, the single player maps can be procedurally generated. This can be very useful since currently a human is needed to create them through some software, and this work is very time consuming. Because of this, a script that generates maps can be very useful for someone to launch a game with a large map pool, or add a lot of replay-ability to a game’s campaign or scenarios. There theoretically should be a group of people who can find a script that does this incredibly useful. Even if the result is too rough to be used as a level in a game, it can save time during testing by allowing testers to test for bugs using a large number of different maps.

To keep the scope of this project realistic, my goal was to create a script which can generate the terrain for RTS maps that can be passable for a stock single player campaign map,
which has enough features to give the player room to make different strategic decisions, as well as determine a good place to spawn the enemy base. In addition, my goal was to create a terrain generator for the Blizzard and Westwood style of RTS, where the terrain itself will have features that affect gameplay. This is because buildings and other doodads are game specific, which could be added to this in the future by modding, and my background knowledge of the other type of RTS games isn’t good enough that I can think of what constitutes a good map for that sub-genre. Also, a goal of this script is not to be able to generate maps that can be used for multiplayer games, as those kinds of maps need to be symmetrical and precisely made to give both players equal footing. I figured creating a script for that will be out of the scope of an honours project.
Methodology

Research

The beginning of the project was comprised mostly of researching different ways of procedurally generating terrain. Before I proposed this topic for my project, I did some research online to see other peoples’ methods on procedurally generating things in general. The first prominent idea I came across is generating a grid, and then manipulating the surface normal of the grid to give it a random and rough texture \(^1\). However, after trying this method for a while I determined it may not be suitable for this since the way the grid is being built isn’t flexible enough for the terrain I wish to build. When I first drafted the proposal for this project, Professor Oliver van Kaick suggested I should start off by looking into using heightmaps. So after some searching I found many resources that explain using heightmaps in terrain generation. Some of these contained example projects which were very helpful, and let me see how many people use the same method to achieve similar results \(^2\). It seemed like heightmaps were the preferred method of many developers wishing to create terrain, so I ultimately decided to use this method.

I had an existing RTS game that was made for a 4th year game development class, so I decided to build my honor’s project off of that game. I was hoping it will save time setting up textures and models, as potentially have the game be playable after the terrain is finished. Another reason for this decision was because the game was developed in the Unity 5 game engine, which provides many built in functions such as easy mesh building. This includes GUI interfaces to adjust variables, easy collision detection, and most importantly, a built in Perlin noise generator function.

Perlin noise is used in generating heightmaps, which are \(n\) dimensional arrays which whose cells contain a number that is used to specify the value of the \(n+1\) dimension at that point. For example, in a 2D heightmap if the value at \((3, 4)\) is 7 then the height value at \(x=3\) and
y=4 is 7 if x and y specify the length and width on the plane. This project will be in the 3rd dimension, so we use a 2D heightmap whose width and length correspond to the X and Z coordinates on our plane, and the values of each cell is the Y or height [3]. Real mountains and terrain tend to smoothly transition from high to low points, so to get a pattern that is similar, we need use something that will give us a random result for the array but also has a gradient between points on the array.

A good solution for this is to use Perlin noise, a type of gradient noise used frequently for making natural looking textures for effects like fire and smoke. The algorithm produces a random noise map with smooth transitions between each node in the noise map, so it will work well for features of mountainous terrain. How Perlin noise works is by first generating a random or pseudorandom vector for each node of the array. Now for any pixel we need to calculate a number for, we can use the four neighbouring nodes’ vectors and compute the dot products of that node’s vector and the vector of that node to the pixel (as shown in Figure 1). With the result of the dot product of the 4 gradient vectors with the vector made from the pixel to the node, we can interpolate those results to assign a number to that pixel. Since the result will only change slightly if the pixel is moved a little, the result is random but it’s smooth.

The following figures give a visualization of the algorithm, the Perlin value at the blue dot is calculated by using the four vectors shown to make four dot product values. This procedure is done for a 2x2 area in this grid to show how the resulting gradient would look like in figure 3, with yellow being a low value and blue being a high value.
In the resources I’m consulting, I’ve noticed many people modify the result of their heightmap using octaves. Octaves are basically more noise that is generated and added to the map. They are represented by a list of vectors with static random values that add/subtract values from each node’s value. Usually this method is very useful to generate terrain that look like mountains, as it the result from using just Perlin noise generally lacks big height changes. Implementing octaves amplifies the height of each node, which fleshes out the features of the terrain to look like mountains with rich features instead of large hills.
Generating the base terrain

In the case of an RTS game, we only want two elevations; a high and low ground that plateaus. So we don’t need to add multiple octaves to exaggerate the results from the Perlin algorithm. Originally I played around with multiple octaves for generating the heightmap, but there wasn’t a visible difference between one or ten octaves. So instead I included just one octave to add some variance to the side of the cliffs and save computing time, but keep the layer of randomness from the octave. This works to amplify each Perlin value to help flesh out the terrain’s features quite a bit compared to without it. As shown in these pictures \[2\], the noisemap on the right uses octaves and has more of a mixed range of values in small areas, resulting in a terrain with more subtle features.

![Figure 6](image1.png)  ![Figure 7](image2.png)

**Figure 6**  **Figure 7**

*Noise maps generated without and with octaves*\[2\]
Visualizations of the noisemaps with and without octaves\cite{2}

To give the script more flexibility I multiplied a value called frequency to the dimensions. This causes the features on the noisemap to change at a different rate, and have the rate of change in gradients to be modified by the frequency value. This is essentially like specifying the wavelength of the map, where if the value is below 1 they will be further, and if they’re above 1 they will be closer together. In this script, since we’ll be modifying the values after the noisemap is generated, changing the value won’t simply change the scale of the generated map; however it does add variance to the end result.

A noisemap generated with and without a low and high frequency value\cite{2}

To save time when generating each map, I restricted the size of the terrain. Also, since I want to move the map around after it’s generated, I added an optional variable to specify the offset of the map. To offset the Perlin noise you simply need to add the offset’s X and Y component to the dimensions being passed into the Perlin noise method. In the code I do this in the equations for the sample X and Y values for the Perlin function.

When combined together, the main chunk of the code that produces the heightmap is represented in the following figure.
The offset vector is a combination of the one octave offset plus the offset vector passed in from the editor. After that I set the default noise scale level to 50 if none is set, and then do a nested loop for all the nodes in the heightmap. Before calling the PerlinNoise method I modify the x and y with the noisescale and then add the offset vector to it. After that I pass the sample x and y to the built in PerlinNoise method, while multiplying the parameters by the set frequency, then doubling to account for the fact that the sample x and y had its original value subtracted by half the terrain’s dimensions. It’s not shown in this screenshot but after some more code to determine the enemy’s spawn location I set the current element in the 2d array to this perlin noise value.

To be able to assign a spawning location for the enemy base, I keep track of the highest point in the heightmap as it is being constructed. Spawning the enemy base at the highest point allows us to be certain it is on the high ground, which is the hardest kind of position to attack in most RTS games. This is not the only way to do it however, we can also assign this location after the map is constructed, so we can scan over the heightmap and find some area with some parameters, such as if the high ground has large enough area, or has certain doodads or other features on it. However since that is not the focus of this project I simply set the enemy spawn location at the highest point in the heightmap, and return the vector so the game can use it.
Another note is although I modified the code for the game to spawn the enemy leader at the location from the script, there is a bug that prevents it from spawning at that location.

Since a large range was used to generate the random numbers, I normalized the heightmap to have all the values be in a range of 0 to 1, based on the values’ ratio to the highest heightmap value generated during the loops. This is done to make adding features and adjusting the final result easier, as we are able to assign the numbers between this range to specific plateaus or elevations. In our case we will want 0 as the low ground level, and some number in between like 0.7 as the high-ground; if we wanted multiple levels of high ground we can split that into other values such as 0, 0.4, and 0.7.

At this point, if one were to run the script it should generate terrain that looks like a mountain range. But since we’re generating a RTS map, we need to smooth out the terrain to have a high and low ground area. Originally I wanted to simply map ranges of values to an elevation in the code. However there is a better and simpler way to do this through Unity’s animation curve object. This produces smoother and more natural looking cliff sides in my opinion, compared to high ground coming to a sudden vertical drop. In addition, this is a variable you can set on the editor via a built in curve editor, which maps some value $X$ to another value $Y$, as shown in figure 11. Here I’ve set values from 0 to 0.45 as the low ground elevation and 0.6 onwards as the high ground elevation.
This approach allows the script to be very flexible for what it’s able to generate. In addition to the accessibility, it’s very easy to visualize the elevations before generating them. This method makes it very easy to adjust how much of each elevation one wants, and if multiple elevations are desired it is also very easy to add them but adding more features to the curve. Getting the mapped values from this curve is also very easy compared to setting specific if statements or switch conditions, since Unity provides a method to pass in a value to the curve and it will give out the mapped value as output. In this project this is done to each element of the heightmap in a nested for loop before other features and added.

```csharp
// Generate the base heightmap
float[,] noiseMap = HeightMapGenerator.GenerateNoiseMap(mapChunkSize, mapChunkSize, RNGSeed, NoiseScale, Frequency, offset, out spawnLocation);
for (int i = 0; i < mapChunkSize; i++)
{
    for (int j = 0; j < mapChunkSize; j++)
    {
        noiseMap[i, j] = meshHeightCurve.Evaluate(noiseMap[i, j]);
    }
}
```
A map creator or modder could use this in creative ways too. For example we can set the elevation at 0 to be some inaccessible area like a body of water, and then have two elevation plateaus above that to be the low and high ground. In addition, it makes it possible to create the types of realistic maps games like Company of Heroes and Rome: Total war uses. Although there needs to be many features to be implemented for a map generator script to be useful for those games, like rivers, bridges, and buildings to be randomly placed, this approach should make it possible to generate terrain for those types of games as well. Figure 13 outlines the low and high ground areas in different colours, as shown in this example there is a roughly even mix between high and low ground areas, but it can be tweaked via the mesh height curve to include more plateau elevations, or the amount of elevation is generated.
Features and doodads

For adding features to this terrain, I used a method similar to how the popular game Minecraft adds its features. The way that game generates its terrain is also by producing the base terrain first, and then adds other features like specific mineral nodes, trees, and etcetera afterwards [5]. It does this by scanning the generated terrain, and when it finds a spot suitable for some feature to be placed, it does a random roll to determine if some feature should be placed there, based on the rarity of that feature [6].

This is handled in a similar way in my project as well. For each feature I want to place, I loop through the 2d heightmap and check if the current location is valid for that feature. The way I determine if a particular area is suitable for a feature is by using a kernel. One of the features I implemented is the tall column with room for a few units, called a pillar. To place this I check all nodes in a square around the current location, and if all of the surrounding terrain is low ground, a random number is generated to see if it should be there. This number is compared to another number set in the editor, and if it’s higher than that number it will place a pillar at that node. To avoid placing too many of these, I keep a list of these pillars that are already on the map, and when checking if a location is suitable for a pillar I check the distance between the current node and the list of placed pillars; if one is found to be too close then it will not be placed. The reason this is done is because relying on the random number generator isn’t good enough to prevent multiple close instances of these. There are rare cases where even if the set threshold that the generated random value needs to pass is high, multiple pillars can still be placed very close or even on overlapping each other.
Figure 16

For the ramps, I originally wanted to ensure each high ground area had at least one ramp by finding all of the high ground locations and inserting a ramp on one of the cliffs. However, I wasn’t able to come up with a good solution for this so I settled by using a kernel that determines if the current location has a cliff. Then using the same method for the pillars, I replace the vertices of that area with a smooth ramp, whose length and width can be adjusted in the map generator editor.
Figure 17 – Ramp with a length of 8

Figure 18 – Ramp with a length of 20
When a node is found capable of having a ramp and passing the set RNG value, I check if that node has a certain side whose mean of the height values on that side is greater than the mean of the values on the opposite side. This is a crude way of determining which way the ramp will face, but it works for creating simple ramps and can be modified to adjust for more directions. After determining which way the ramp ascends I create the ramp with the selected node as the middle of the ramp. I could have used the node as the top, or near the top of the ramp but I like the ramp merging into the cliff’s side as it looks smoother.

Lastly, the heightmap is passed into a method which returns a mesh to be assigned to the terrain. The method loops through all of the nodes in the heightmap, but increments based on the LevelOfDetail variable set in the editor. When the increment is higher the triangles will be larger, resulting in smoother looking terrain. In the loop I simply create two triangles and set the texture coordinate for that index. This mesh is applied to a GameObject on the scene, which can have other things added to it like colliders and effects.

To make modifications to the generated result easier, I created a Unity editor for the script. It can be applied to any GameObject on the scene, and allows the variables with the public keyword on the MapGenerator class to be edited via a GUI window.
This is one of the other advantages of creating this script in Unity, it is very easy to make a GUI like this, and it’s even possible to generate multiple terrain at once by including other instances of this script in the current game scene. As a side note, all of the screenshots in this report are taken with these values as settings.
I am fairly satisfied with the end result of the project. It achieves almost all of the goals I originally set out to do, with the exception of some bugs. The script provides an editor for some user the ability to manipulate variables to generate different looking terrain. The code of the script is also designed in a way that modifications and addition of other features is fairly easy, as you just need to create a separate method that specifies the location of certain features and add them before the mesh curve is applied.

Implementing Perlin noise and smoothing out the result with the curve map results in a satisfactory placement of high and low ground for RTS maps. Modifying the variables for the Perlin noise function also makes the cliffs a bit more refined. The user is also able to specify at which elevation the high ground starts, so it’s possible to have more high ground and change the amount the cliffs drop as they see fit by modifying the animation curve.
This script was created with scalability in mind, so adding features to it is fairly easy. All that needs to be done is create a method and call it in the Generate method in MapGenerator.cs. In addition, I’ve included adjustable variables for as many things in this project, which allows for flexibility when generating maps.

For example, one could create impassable terrain by creating an animation curve with a steep drop from 0 to 0.1, then have a plateau at 0.1 and 0.5. Then when the mesh is generated, a different color, like blue can be set at those vertices to represent water. After that the user can add a mesh collider at that area that prevents units from going inside the area. Another example of a feature that can be implemented is a chokepoint with a clearing after it, which can be implemented by doing a roll at a high ground area’s cliff, then if it passes using the first point make a small circle of terrain low ground, then for some length, make transform larger circles of
high ground into low ground terrain, so you would end up with a small opening at the side but a cone shaped low ground section after it.

There are some issues left in this project, but due to time constraints I was not able to fully fix them in time. The biggest issue in my opinion, was being unable to determine the why units from the game itself is not able to collide properly with the mesh collider on the terrain. The mesh generated renders in the correct orientation, and lighting effects on the texture implies the normal are set correctly; but even with both GameObjects having colliders units will phase through the terrain once off the initial grid. Similarly, I think this also causes the enemy command unit to not spawn at the location given back from the terrain generator script. I’m not completely sure if that bug is caused by the same reason, but I highly suspect that is the case since I was able to spawn the enemy leader in the original game this script is built on.

In addition, there are a number of bugs with the current build. The placements of ramps needs further refinement as placing them completely randomly sometime causes it to be in places that aren’t exactly on the edge of a cliff. These errors are mainly a result of my lack of experience with Unity; especially with collisions as in previous projects other team members handled collision detection. Although troublesome, the core features of this script is still working as intended and I will continue to try to resolve these bugs.

As shown in the figures, through the unity editor one is able to save their generated content by saving the RNG seed used, and the values of the variables. If one does not necessarily want the entire script used in the game and just the map itself, the mesh can be exported after generating the map in another project. This allows potential indie game projects to use the script in different ways, to create maps for scenarios or even test different kinds of maps without requiring a person to explicitly create maps.

The end result of this project ultimately achieves what I originally set out to do. Although not usable with the other game objects in the scene currently, it is able to generate a map that looks similar to generic maps from Starcraft and Command and Conquer games. I hope this script will prove helpful to developers who have an interest or are working on RTS
games. My intent is to continue working on the script, so more features can be added and the aforementioned bugs can be ironed out.
References

Appendix

Implementation notes
It’s possible at this point to just run the Mathf.PerlinNoise() method after getting the sample x and y values for the method. This would result in an easier implementation; but it’s possible to modify the results by changing the dimensions being passed into the noise function in other ways than just adding the frequency. If one desires, it should be possible to skew the results from the Perlin noise function by multiplying other values for example, to unevenly scale the results. I didn’t have use for this in my implementation but for someone else it may be useful.