Modern Game AI Algorithms

Practical Assignment 1: Procedural Content Generation

N. Witte
s2685620

March 11, 2021

https://github.com/Trottero/game-ai-1

1 Introduction

This report summarizes the work done by Niels Witte for the first assignment of the Modern Game AI and Algorithms Course. This assignment, titled Procedural Content Generation, provides an introduction to concepts of generation of content which are used in games such as Minecraft and No Mans Sky.

2 Background

This section will provide an introduction to techniques used to generate randomized procedural generated maps.

2.1 Procedural Content Generation

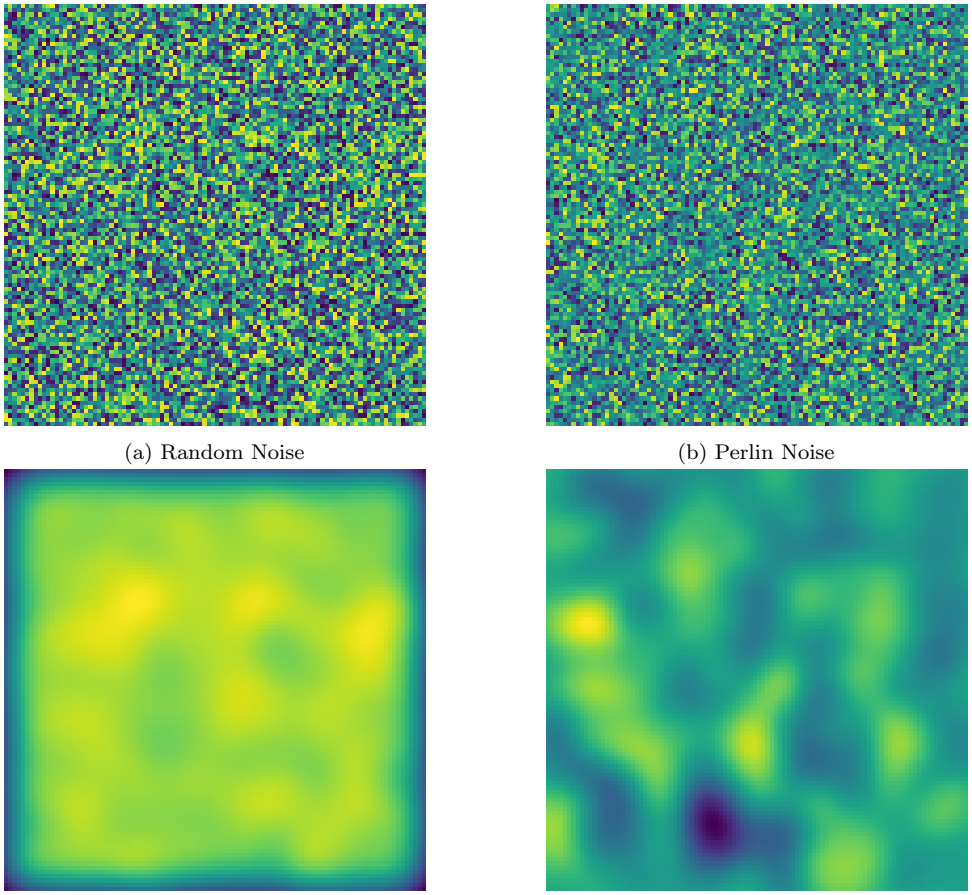
Procedural Content Generation is the generation of additional content in such a way that it can always be expanded upon and stopped at any time.

Believability A key factor is that one piece of content should flow into the next piece of generation of content. When generating terrain for example, it is expected that a transition from a grassy area to an ocean would not only be gradual in terms of height but also in terrain type which can be achieved by adding a beach layer between the grass and the ocean. This also translates to land features such as mountains and plateaus. Mountains are often clustered together with the highest peak being somewhere in the centre of the cluster.

2.2 Algorithms

Perlin Noise For height generation, Perlin noise (Perlin, 1983) is used. This is a semi-random noise generation algorithm which has seen widespread use in generation of content for games but for different applications such as generating fake clouds for movies. It works by generating random vectors on a given grid and then interpolating the tiles around the generated vector. Figure ?? compares the complete random noise and Perlin noise. Although it might not instantly obvious that there are some patterns involved it becomes clear when either multiple layers are stacked on top of each other or when a Gaussian filter is applied as in Figure 1.

Dijkstra's Algorithm Roads are a part of the content that we are planning to generate and in order for the roads to be believable the fastest route is calculated and drawn between towns. Like in real life, some terrain like water or woods are harder to cross than the grasslands. Dijkstra's



(a) Random Noise (b) Perlin Noise

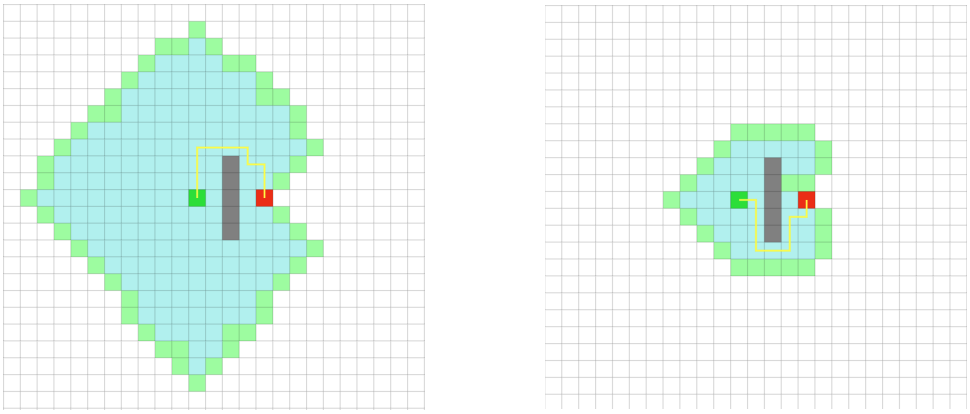
(c) Random Noise with Gaussian Filter (d) Perlin Noise with Gaussian Filter

Figure 1: Comparisons of random noise to Perlin Noise with and wihout applying a gaussian filter. Seed: ‘banana‘

algorithm (Dijkstra, 1959) tries to find the path with the lowest cost between two points, in our case towns. It does this in an explorative manner where it automatically explores the area around a given point in a breadth-first manner.

A* Dijkstra's breadth-first search method causes it to explore the tiles around the starting tile in a circular fashion, which means that it'll also explore tiles which actually move away from the objective. This behaviour would be desired if we didn't have access to the targets position. In our case where we will use Dijkstra to generate an efficient road between towns this is however not the case. A* (Hart, Nilsson, & Raphael, 1968) is an extension on Dijkstra's algorithm where an Heuristic is used as a part of the cost function for a given tile. This effectively forces Dijkstra's algorithm to minimize a combination of both the current tile cost as well as the heuristic, which turns this into a somewhat directed search. Figure 2 compares Dijkstra's breadth-first behaviour to A*'s directed behaviour. This figure shows that A* visits significantly less nodes than Dijkstra.

Kruskal's Algorithm Our network of towns can be seen as a fully connected graph. Since connecting every town with every other town would clutter the map with unnecessary roads it is important that some roads which don't make sense are removed. For this we represent our towns as vertices in a fully connected graph. We then have several different options, we can randomly start deleting edges as long as they don't disconnect a set of vertices from the main graph or we work



(a) Dijkstra traversal graph

(b) A* traversal graph

Figure 2: Comparison between Dijkstra's Algorithm and A*. Source is green, target is red. Blue squares are visited nodes with light green being the next nodes to potentially visit (Xu, 2011).

the other way around where we progressively add more edges to the final graph until all of the vertices are connected. Kruskal's (Kruskal, 1959) algorithm follows this exact procedure by first sorting all of the edges by their weight and then progressively adding them to the final graph until all vertices are connected. This results in what is known in graph theory as a minimal spanning tree and will form the road network that is presented in our map.

3 Implementation

In order to create our own procedurally generated map the following features have been implemented.

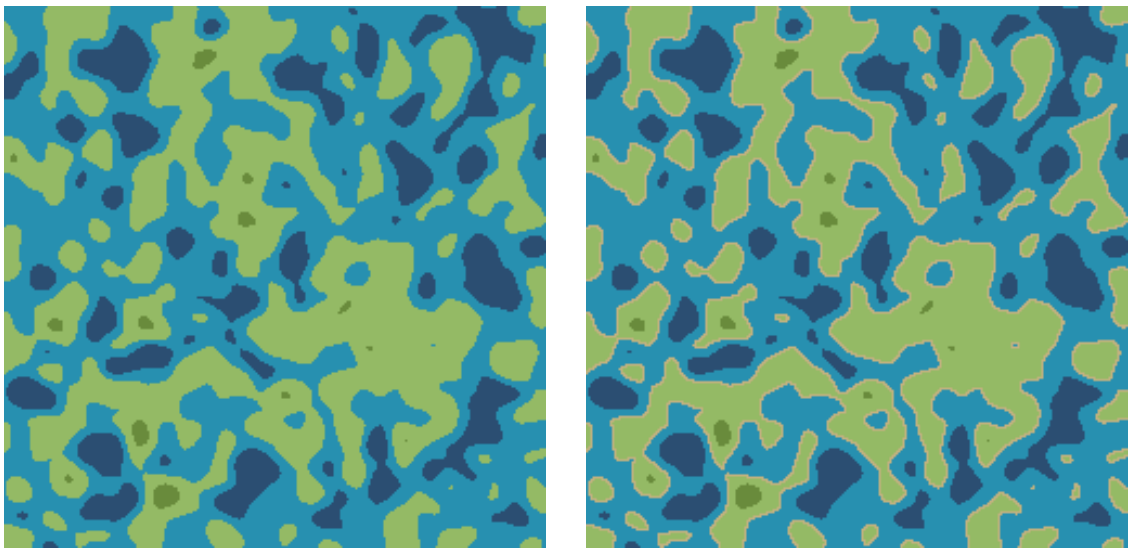
- Height based terrain
- Beaches
- Towns
- Road network

These were implemented in a python program with every individual feature as a separate generator layer which can then be layered together to create a fully fledged game map. This implementation should be easily extensible so layers can easily be swapped out, configured and added upon.

Height based terrain is implemented using a combination of Perlin Noise with a Gaussian filter applied to it. This is then stored in a normalized height map. This map is then converted into a world map by assigning specific tile types to the given values. Everything above 0.5 is considered land with values over 0.8 being dense woods. Values under 0.5 are considered water with values under 0.2 being considered oceans. Note that beaches are not included here as this would cause random spots of beach to appear in low parts of the grasslands. Applying this generator results in Figure 3.

Beaches In order to implement beaches several approaches can be considered; we could for example generate another Perlin Noise map and use that in combination with our existing height map; But I instead chose for a more simple and consistent method. It involves a basic kernel which

is dragged over the entire map. This method was inspired by convolutional layers in Convolutional Neural Networks. It drags a matrix of a predetermined size and subtracts a given number from all of the tiles in the area. This results in an x amount of zeros in the resulting matrix which can than be counted for the amount which represent the amount of water tiles in the given area. The same is then repeated for the land tiles. This way beaches wont appear inland nor in the middle of the ocean. If these values meet certain configurable thresholds, a land tile is converted into a beach tile. This results in extremely consistent beaches as can be seen in Figure 3. By increasing the kernel size or decreasing the thresholds wider beaches can be generated.



(a) Result of applying the land generator

(b) Result of applying the beach generator

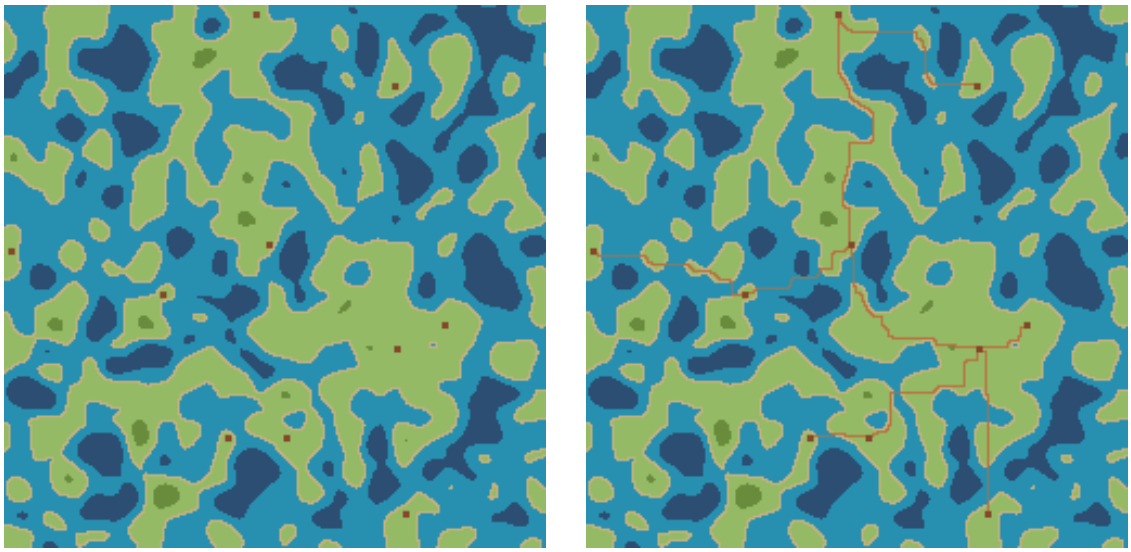
Figure 3: Resulting world maps after applying the land and beach generators. Seed: 'blueberry'

Towns The generation of towns or villages is a fairly simple problem and can be formulated as; For every $n \times n$ chunk of land, a town could appear with probability P_t. This is fairly straightforward to implement, we split the map into $n \times n$ chunks and determine using P_t whether or not an attempt at spawning a town should be made. If it is determined that this chunk should have a town, up to 10 (configurable) spawning attempts are done by randomly selecting coordinates within the chunk and checking if the 2×2 area is only grass. If this is the case, a village is spawned. This results in evenly spread villages all across the world map which can be seen in Figure 4.

Road network The Road network is one of the most complex problems of this assignment. As outlined in subsection 2.2 there are multiple ways to generate a network but Kruskal's algorithm is picked here. In order to execute this algorithm all of the edges of graph (our fully connected town network) need to be collected. This means that every possible path needs to be pre-calculated before executing the algorithm. This is actually quite an expensive operation as the number of edges is directly related to the amount of vertices in the graph and follows the formula $\frac{n(n-1)}{2}$. Therefor this operation is being multi-processed with a 16 core CPU and even then generation of a single 250×250 world map takes over 5 minutes to generate roads on.

The resulting information is used to run Kruskal's algorithm which results in Figure 4. Here we see the impressive path finding behaviour of A* where it clearly prefers bits of land over the water it encounters on its way. Both the west and north east portions of the map show an island hopping

behaviour where it clearly avoids as much water as possible. This behaviour is created by assigning different costs to different types of land. Grassland and beach have a cost of 1 and 2, where as woods, ocean and deep ocean have costs of 5, 20 and 35 respectively.



(a) Result of applying town generator

(b) Result of applying road generator

Figure 4: Result of applying the town and road generators to the previous world map. Seed: ‘blueberry‘

Diversity Maps can be generated using a given seed where the same seed provides the same map every time. 6 different seeds; ‘banana‘, ‘dragonfruit‘, ‘grapefruit‘, ‘blueberry‘, ‘watermelon‘ and ‘papaya‘ were used. All 6 seeds resulted in reasonable world maps but blueberry (Figure 4) and banana (Figure 5) were handpicked because of their interesting roads around water and their large landmasses.

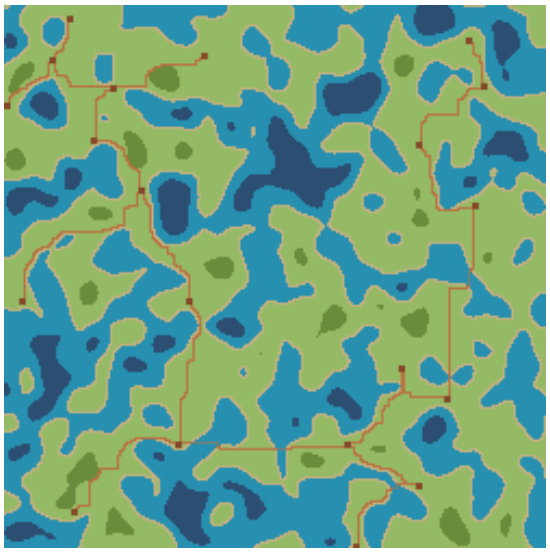


Figure 5: Fully generated world with the seed ‘banana‘

4 Conclusion

In this assignment I've shown that by using Perlin Noise, Beach Kernels, Dijkstra, A* and Kruskal's algorithm a fairly simple world map can be procedurally generated. In order to promote believability, terrain with varying height has been implemented with the addition of grass, woods, ocean and deep ocean. Beaches were also added as a transition layer between grass and water. The world map was decorated with towns which are all form a connected network together. Interesting behaviour of A* was observed and an easy to extend python implementation was provided. This implementation might not be that performant, but maps up to size 250 can be generated in a reasonable amount of time which is plenty for this assignment.

Future work could include the addition of a more diverse height map, where actual gradients of green are used to differentiate between high and low grasslands. It could also include different biomes with different generation, such as a desert with less hills or a badlands biome with rivers running through the rocks or just adding simple features such as rock outcrops, coral in oceans or rivers would also make great additions and make the world more lively. I personally think that adding small details to a world makes it significantly more believable.

On a final note I'd like to say that I really enjoyed this assignment as it served as a creative outlet within programming without having to pick up a pencil or drawing tablet. The art of creating a virtual environment is quite interesting to me and the only limitation is ones imagination.

References

Dijkstra, E. W. (1959). A not on two problems in connexion with graphs. In *Numerische mathematik* (pp. 269–271).

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100-107. doi: 10.1109/TSSC.1968.300136

Kruskal, J. B. (1959). On the shortest spanning subtree of a graph and the traverling salesman problem. In *Proceedings of the american mathematical society* (p. 48-50).

Perlin, K. (1983). *Kevin perlin's original perlin noise implementation.* Retrieved from `https://mrl.cs.nyu.edu/~perlin/doc/oscar.html#noise`

Xu, X. (2011). *Pathfinding.js: A comprehensive path-finding library in javascript.* Retrieved from `https://github.com/qiao/PathFinding.js`