Modern Game AI

Final project

Matthias F.A. Aarnoutse	Benjamin Havenaar	Job M.C. Vink	Niels Witte
s2682796	s3026531	s1869205	s2685620

May 20, 2021

1 Introduction

When creating a realistic and immersive experience for players, the surroundings, interactions with the world and other characters are of vital importance. In this work we explore various procedural content generation methods, realistic non-player behavior techniques and methods to efficiently produce realtime game content.

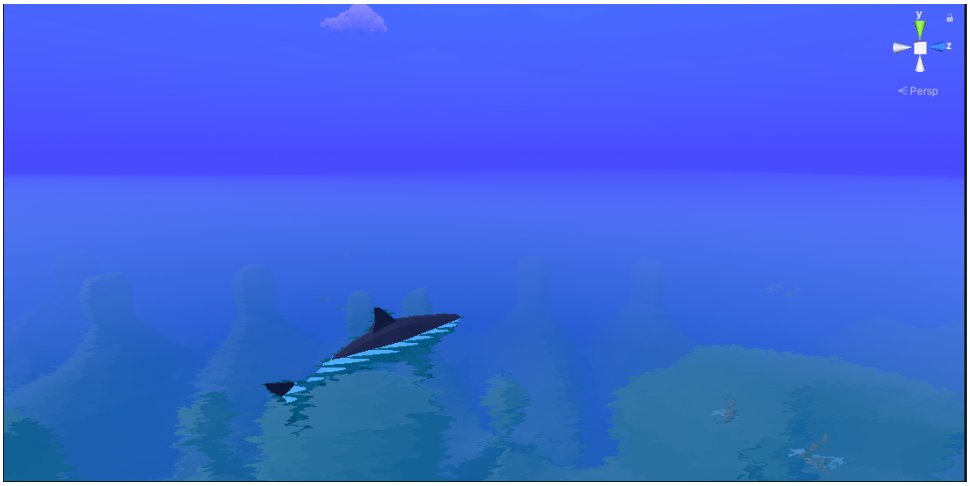


Figure 1: *Above water view with our player shark and animated water. A couple of fish can also be spotted in the bottom right of the picture.*

We create an underwater environment with a sea floor and rocky cliffs using the marching cubes algorithm. We use blue fog with a Perlin Noise mesh for creating the water and surface. To create an infinite terrain we use a chunking method to procedurally generate new blocks of terrain based on player movement.

The underwater world contains of randomly generated fish that follow simple individual flocking rules to behave as a cohesive school of fish. The fish can individually track predators and objects within their perceptive field. The player is a shark that can swim between the fish.

The goal of this simulation is to create a realistic infinite underwater world with a non-player fish following realistic behavior and capable of interacting with its surroundings and the player.

2 Procedural Terrain

Taking inspiration from games such as *No Man's Sky* where every planet is generated, we set out to create a procedurally generated ocean. As a first step we used Perlin Noise to generate terrain (Figure 2). However, this did not create the effect we were looking for as the ocean bed would be quite 'flat'.

We then came across another algorithm called 'Marching Cubes' which seemed as the perfect algorithm for the job. The generation of the terrain is based on a single density function that describes for a single

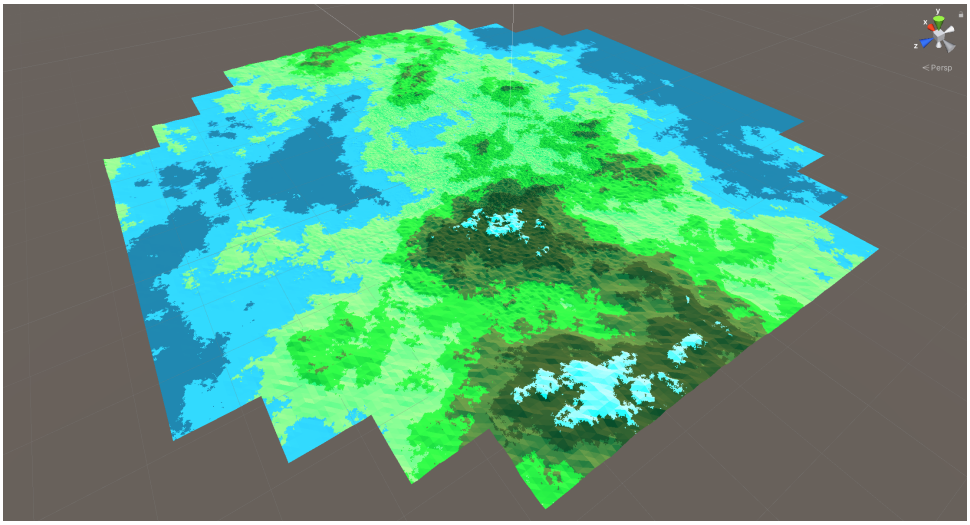


Figure 2: *Terrain generated with Perlin Noise*

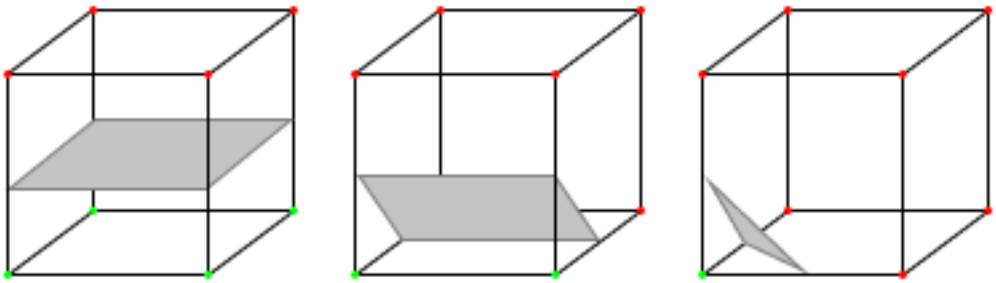


Figure 3: *The result of the marching cubes algorithm visualized, the corners marked in red hold negative density values and the corners marked in green hold positive density values.*

point in 3D space (x, y, z) if that point in space is inside the solid terrain or not $D\colon \mathbb{Z}^3 \to \mathbb{R}$. Values higher or equal to zero indicate that the point is inside solid terrain. Values lower that zero indicate that the point is outside solid terrain.

$$T = \text{sign}(D(x, y, z)) \tag{1}$$

2.1 Marching cubes

We can use the density function D to generate a block of terrain. By utilizing the marching cubes algorithm (Lorensen & Cline, 1987). By further dividing the block into cubes we can generate the polygons needed within a single cube given the density values at its eight corners. The algorithm works as follows. Given a block divide the space into an arbitrary number of cubes. Test the corners of every cube for if they are inside solid terrain or not. Intersect the edges of the cube in between corners of opposite classification, exactly at the midpoint relative to the amplitude of the corner values. Draw the surface within each cube connection these intersections. How this algorithm results in a surface is visualized in Figure 3.

Density function To make a realistically looking terrain we use perlin noise generation (Perlin, 1985). This algorithm generates random vectors for each point that influence the final result rather than taking purely random values. To generate a seafloor we want to confine the density function to a certain height in the 3D space. This is done by penalizing the density if it exceeds certain limits for the top and bottom.

$$D(\vec{p}) = G(\vec{p}) - C|y - h_t| + C|y - h_b| \tag{2}$$

Where G is the noise generated by the Perlin Noise algorithm as point p. C is a tunable parameter indicating the magnitude of the penalty. h_t Is the height of the coordinate and h_t is the height as which the seafloor should stop and h_b is the height at which the seafloor should be solid.

In order to create even more complex terrain multiple layers of noise can be combined. Noise with a low variancy and high amplitude dictates the core shape of the terrain. This can then further be enriched by adding another layer with high variancy but with a low amplitude on top. This gives texture to hills and transforms the landscape from dunes into a rocky landscape.

It is also possible to apply a bias in this density function. This often takes the form of a small value which is added or subtracted from the final density value. Higher values result in higher densities and thus in more terrain, effectively increasing the density of a specific portion of the terrain. We use this principle to create different densities which opens up caves without opening too much on the surface.

2.2 Infinite Terrain

According to Schneider, Root, and Mastrandrea (2011) water covers approximately 75% of the earth's surface. Containing the fish in a finite space would therefore not make sense because from the fishes perspective it could feel like there is nothing else than water. To simulate this infinite space of water brings some technical challenges. Since the terrain is infinite it is impossible to generate it beforehand as this quite literally would take forever. Instead the terrain is divided into chunks.

We can achieve this by saving the players position $\mathbf{p}$ and defining a maximum visible distance d. If the the delta between the current and previously saved position exceeds the threshold, new chunks can be generated. The current position is then saved as the previous. This makes the computation less intensive as the chunks are updated less frequent. The infinite terrain generation is demonstrated in Figure 2 where chunks have been generated in a radius around the player.

$$||\vec{p}_t - \vec{p}_{t+1}||^2 > d^2 \tag{3}$$

This opposes a new problem however, if you keep generating new chunks of terrain you will eventually run out of memory even on modern hardware. This is why we remove previously generated chunks that are now out of range. These will be generated once they are back into the view. This is possible due to the fact that we use a deterministic terrain generation algorithm. The density function D mentioned in Section 2.1 only takes the coordinates of the generated block as its parameters. The terrain can be continuously generated and regenerated with the same result. Only the vertices at the border of the block have to be connected to the newly generated block by using the marching cube algorithm.

In order to hide the fact that new chunks are loaded dynamically we introduce a fog effect which limits the player's range of vision. The distance that this fog is rendered at; d, is slightly less than the generation range, this gives the player the illusion that the terrain has always been there.

3 Flocking

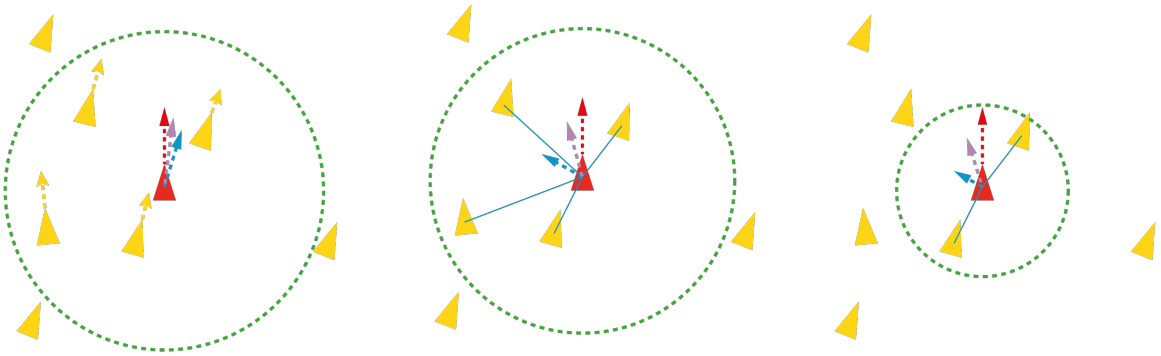
3.1 Natural behavior

Birds, fish and other herd animals are capable maintaining a cohesive pack by each individually adjusting according to a set of simple rules. Without a distinct leader or set out path, a herd can navigate its

surroundings while staying in close proximity to eachother. Flocking animals can make complex and seemingly coordinated moves together. One reason for this behaviour is increased survivability from predators. In the case of fish, a large enough school has a full 3D view of it's surroundings making detection of predators easier. Also the survivability in terms of the chance of being victim to the predator for an individual fish is lower while swimming in a school. This increased perceptive field in a school also positively benefits foraging for food. Finally schooling makes finding a mating partner easier which increases the probability of reproduction. (Pitcher, 1986)

Not every fish type schools the same. A distinction can be made between obligate schooling fish such as tunas, herring and anchovies. And on the other hand facultative schooling fish such as coal fish and cod. Obligate schooling fish are always found in schooling formation with fish from the same species or similar fish. Facultative schooling fish swim primarily alone but in some occasions as a school. (Breder, 1967)

A school of fish can have different short term goals such as feeding, traveling or fleeing from a predator. Each of those short term goals have other requirements to the tightness and and strictness of the school. When going from one goal to another fish can rapidly change from one schooling behavior type to another. (Pitcher, 1986)



(a) ***Alignment:*** *the red boid looks at the current direction of it's neighbours (yellow boid) that are within the receptive field and calculates an average direction vector (blue-dotted arrow).*

(b) ***Cohesion:*** *the red boid looks at the vector distances (blue lines) of its neighbors (yellow boids) and calculates the average vector (blue-dotted arrow) that points closer to the neighbors*

(c) ***Separation:*** *the red boid looks at the vector distances (blue lines) to it's neighbors (yellow) that are within the receptive field and calculates the inverse of the average vector (blue-dotted arrow) that points away from the neighbors.*

Figure 4: *2D representations of the effects of alignment, cohesion and separation. The green dotted line represents the receptive field each rule. The red-dotted arrow is the current velocity vector v_t, the blue-dotted arrow is the direction vector of the particular factor, and the purple dotted arrow is the new vector v_{t+1} if considering that trajectory influencing factor.*

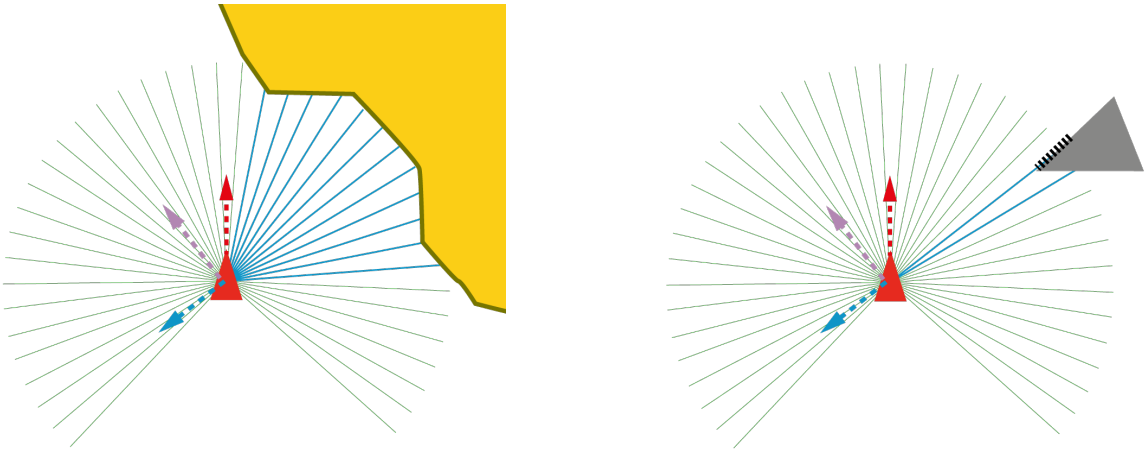
3.2 Boid interaction

As proposed by (Reynolds, 1987), flocking behavior can be modeled by computers using simple rules for each different bird, fish or boid to follow. The direction and velocity of the boids are influenced by various perceptions of it's surroundings. Each perception has it's own influence the direction vector of the next step v_{t+1}. Figure 4 shows the interaction between the boids that are distinguished in three behaviors and calculated each frame for each boid. The *alignment* (Figure 4a) is calculated by taking the average alignment direction vector v^a of the neighboring boids within the alignment perceptive field. The *cohesion* (Figure 4b vector v^c is calculated by evaluating the distances between each neighbor within the cohesion receptive field. The *separation* (Figure 4c) vector v^s is calculated the same as the cohesion vector but then by taking the inverse. For each behavioral element a different perceptive field is determined. While cohesion and separation are inverse operations, taking different perceptive fields can make the boids avoid near neighbors while aiming for the flock center. Dependent on the tightness of the flocking

conditions, fish can also break loose from schools and swim alone until it meets another school. This way we are able to model the extent at which fish are obligate schooling or facultative schooling.

3.3 Collision avoidance

The collision avoidance is done by creating a different type of perceptive field for each boid. Whereas boids 'know' each others exact position from memory (and thus if the boid is within the perceptive field). The boids do not know each coordinate of the surface of the objects that need to be avoided. Avoidable objects are the terrain boundaries such as the trough, seabed, water limit) and a predator.



(a) ***Surroundings:*** *the red boid looks at objects in it's surroundings. The yellow surface represents a rock object what the red boid attempts to avoid.*

(b) ***Predator:*** *the red boid looks at predators in it's surroundings. The grey boid represents a shark that the red boid attempts to avoid.*

Figure 5: *2D representation of collision avoidance and the receptive field. Green and blue lines are rays that are evenly distributed over a curvilinear receptive sphere. The blue lines are the rays that collide with an object. The red-dotted line with arrow is the current direction of the boid, the blue-dotted line with arrow is the average vector of the collided rays finally the purple line is the combined vector of the current direction factor and the collision avoidance vector without consideration of other influencing factors.*

As the 2D representation shows in Figure 5 each boid has a curvilinear receptive field. We use a point distribution method to evenly distribute 50 perception rays on that sphere. If a terrain object comes within the perception range the inverse average avoidance vector v^e from the collided rays is calculated. The same technique is used for the shark with the vector v^p.

Finally the optimal vector is calculated by multiplying all boid interaction vectors and the collision avoidance vectors with their weight factors.

$$v^o = v^a q^a + v^c q^c + v^s q^s + v^e q^e + v^p q^p \tag{4}$$

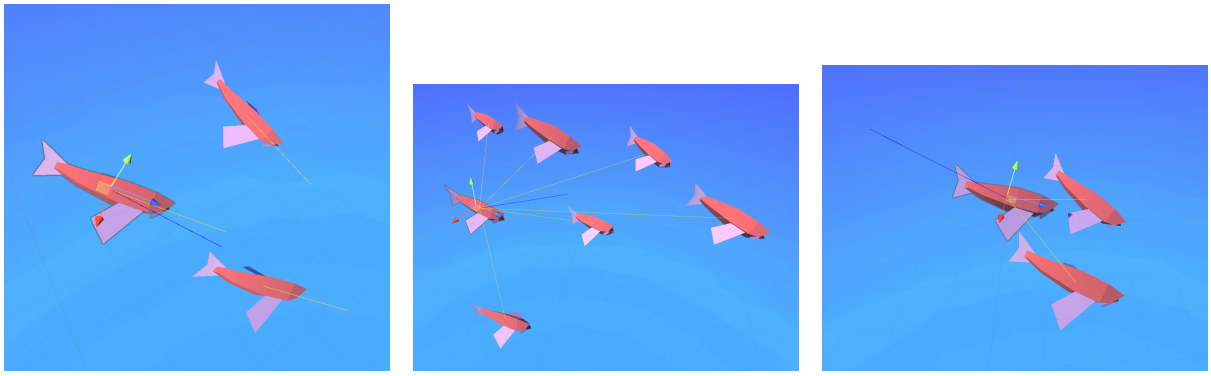
Once the optimal vector v^o is calculated we can calculating the new vector v_{t+1} by rotating v_t towards v^o within a maximum turning speed of 2.5 radians (143.2 degrees) per second.

4 Implementation

The entire project was build using Unity which is a game engine in which one can script using the C# programming language. For highly performing post processing effects we use shaders written in a

language called HLSL. Some animations such as the waves in our ocean and the water are implemented with these shaders.

The results from Section 3 'Flocking' are displayed in Figure 6.



(a) ***Alignment:*** *the selected fish looks at the current direction of its neighbours that are within the receptive field and calculates an average direction vector (blue line).*

(b) ***Cohesion:*** *the selected fish looks at the vector distances (yellow lines) of its neighbors and calculates the average vector (blue line) that points closer to the neighbors*

(c) ***Separation:*** *the selected fish looks at the vector distances (yellow lines) to its neighbors that are within the receptive field and calculates the inverse of the average vector (blue line) that points away from the neighbors.*

Figure 6: *In-game representations of the effects of alignment, cohesion and separation. The green dotted line represents the receptive field each rule. The red-dotted arrow is the current velocity vector v_t, the blue-dotted arrow is the direction vector of the particular factor, and the purple dotted arrow is the new vector v_{t+1} if considering that trajectory influencing factor.*

The finished product is displayed in the following figures; Figure 7 demonstrates the fog that hides the edge of the viewing distance, Figure 8 shows the terrain generation underground, and Figure 1 displays the beautiful distortions in the water.

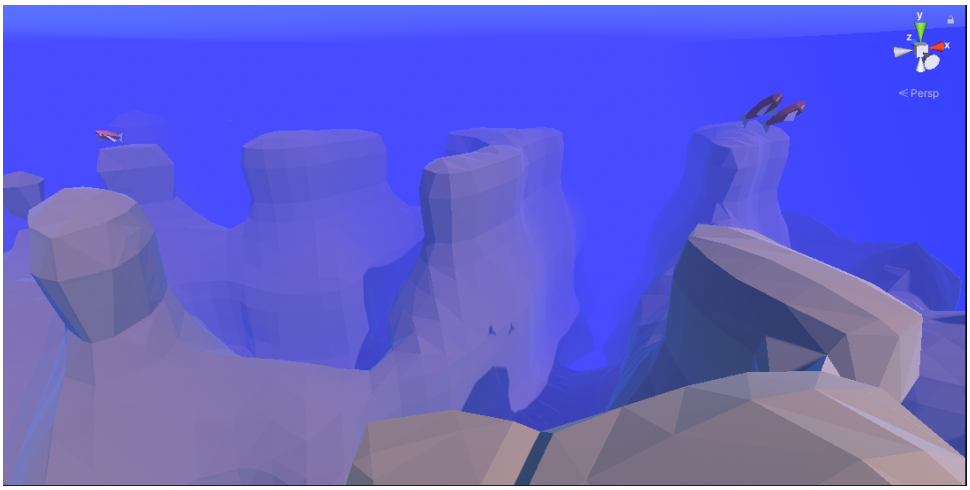


Figure 7: *Fog enabled on our final configuration. The edges of the terrain are hidden from view and a blue tint is added to the environment.*

We experimented in an empirical manner on the terrain generation parameters, these results are displayed in Appendix A.

4.1 Benchmarking

To be able to measure the performance in the game we introduced an FPS counter (Figure 9).

Figure 8: *Underground view of our final configuration with fog enabled.*

Figure 9: *FPS Counter in-game on 4K*

The results of our benchmark are displayed in Table 1. They have been run on an Nvidia GeForce RTX 3070. It is clear to see that 'FPS Low' is not a great indicator as all configuration receive 3 fps as their lowest. This is likely due to the fact that the game is taking a long time starting up. The average FPS shows the best correlation between the resolution and performance.

Resolution	FPS High	FPS Low	FPS Avg
3840×2160	188	3	34
2560×1440	247	3	41
1920×1080	212	3	92

Table 1: *Performance on different resolutions*

5 Conclusion

In this work we explored various techniques for procedurally generating underwater world. We found that using Perlin Noise was not a suitable algorithm to create an interesting coral like ocean bed. By using the marching cubes algorithm with a density function which stacks multiple layers of noise we were able to create a more realistic terrain. In order to find the best parameters for the density function we carried out experiments where we empirically found a set of parameters which created a believable environment.

In order to create the feeling of having an infinite ocean to explore we generate terrain blocks in segments and apply a light fog effect to mimic seawater. This allows us to (re)create and destroy terrain in real-time as the player moves through our landscape.

In order to make the environment come to live we introduced schools of fish which implement the 3 rules of flocking and our results are shown in Figures 6a, 6b and 6c.

In the future we would like to increase the performance of terrain generation as the generation of new blocks causes frequent stutter in the application. We also think that it is worth to look at a specific raycast mechanic for the predator, this would allow finer control over the behaviour of the fish when they encounter said predator. Finally we would like to introduce flora such as coral and seaweed to our environment.

Overall this was a very interesting project for us. Real-time environment simulation is quite a complex task and can require a lot of compute power. Before starting this project none of us had used Unity before which also slowed us down quite significantly. We are however quite pleased with the result and under the impression that we have created a visually pleasing and believable underwater environment.

References

Breder, C. (1967). On the survival value of fish schools. *Zoologica; scientific contributions of the New York Zoological Society*, *52*, 25-40.

Lorensen, W. E., & Cline, H. E. (1987, August). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, *21*(4), 163–169. Retrieved from `https://doi.org/10.1145/37402.37422` doi: 10.1145/37402.37422

Perlin, K. (1985). An image synthesizer. In *Proceedings of the 12th annual conference on computer graphics and interactive techniques* (p. 287–296). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/325334.325247` doi: 10.1145/325334.325247

Pitcher, T. J. (1986). Functions of shoaling behaviour in teleosts. In T. J. Pitcher (Ed.), *The behaviour of teleost fishes* (pp. 294–337). Boston, MA: Springer US. Retrieved from `https://doi.org/10.1007/978-1-4684-8261-4_12` doi: 10.1007/978-1-4684-8261-4_12

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on computer graphics and interactive techniques* (p. 25–34). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/37401.37406` doi: 10.1145/37401.37406

Schneider, S., Root, T., & Mastrandrea, M. (2011). *Encyclopedia of climate and weather* (No. v. 1). OUP USA.

A Terrain Generation

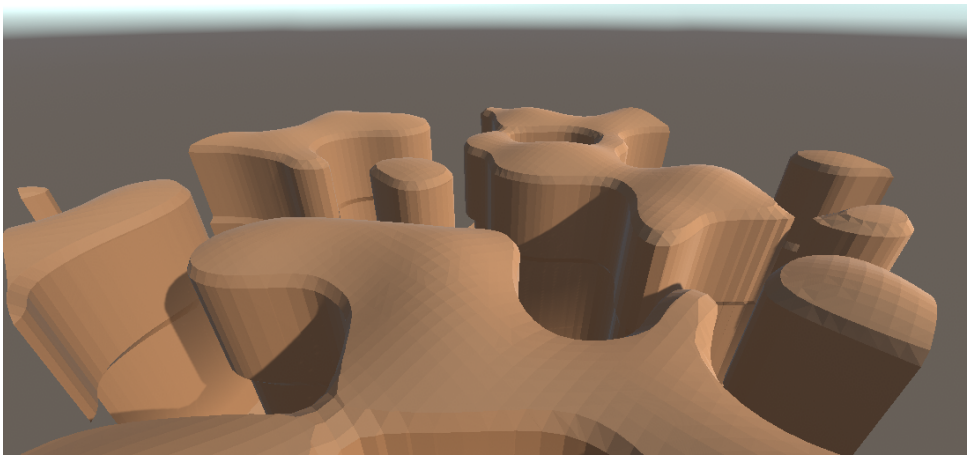


Figure 10: *Density function with just horizontal noise, with an equal variance and amplitude of 1.*

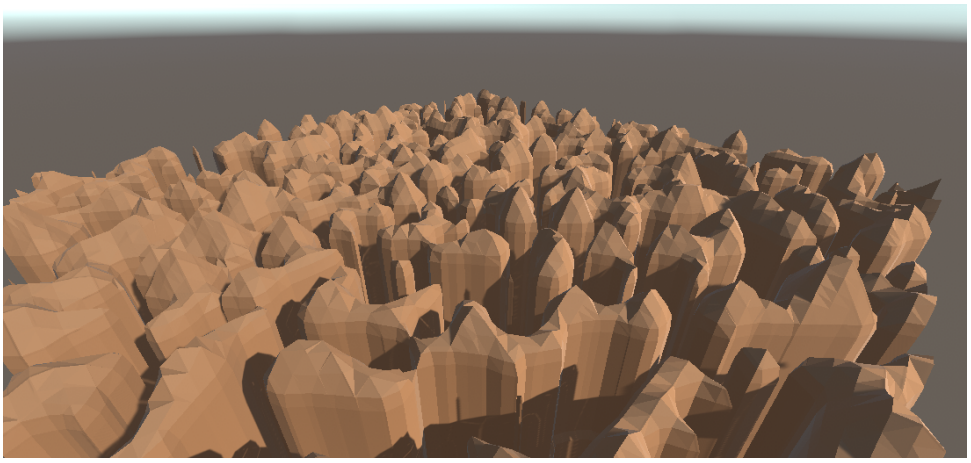


Figure 11: *Density function with just horizontal noise, with a high (4) variance and regular amplitude of 1.*

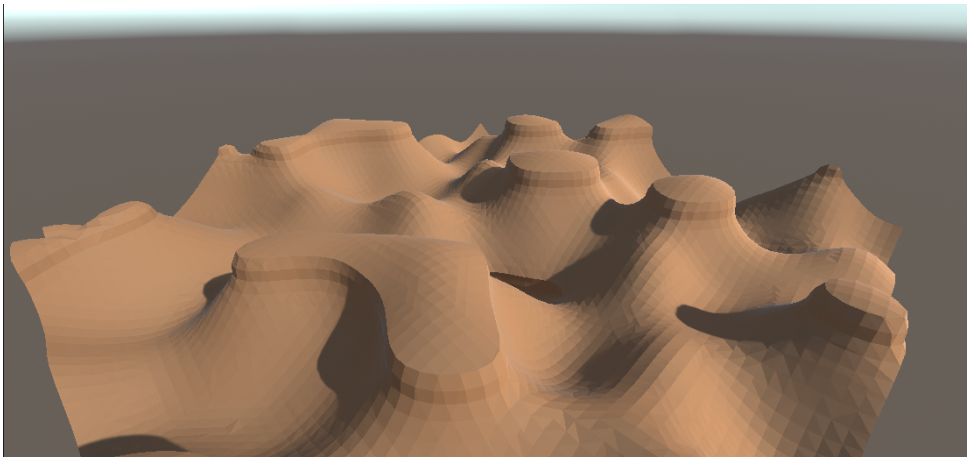


Figure 12: *Density function with horizontal and vertical noise combined, with variance and amplitude being 1 for the noise.*

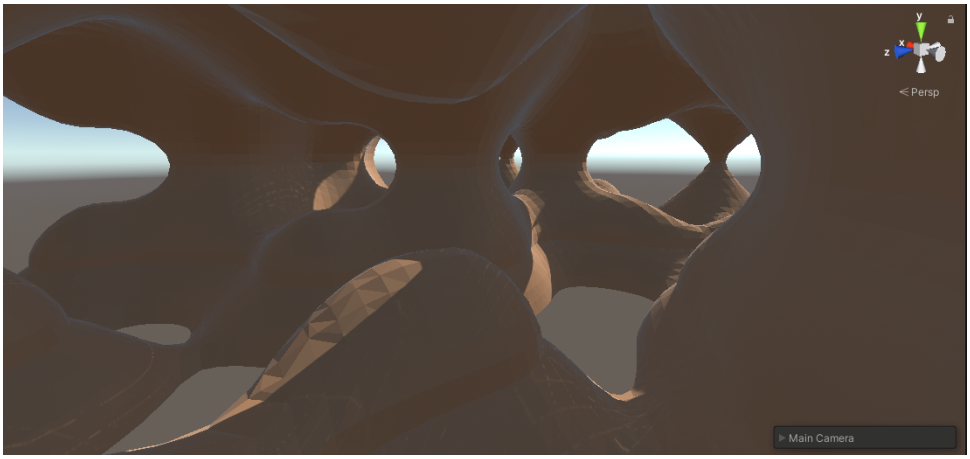


Figure 13: *Density function with horizontal and vertical noise combine, with variance and amplitude being 1 for the noise underground. We can see that basic caves start to form.*

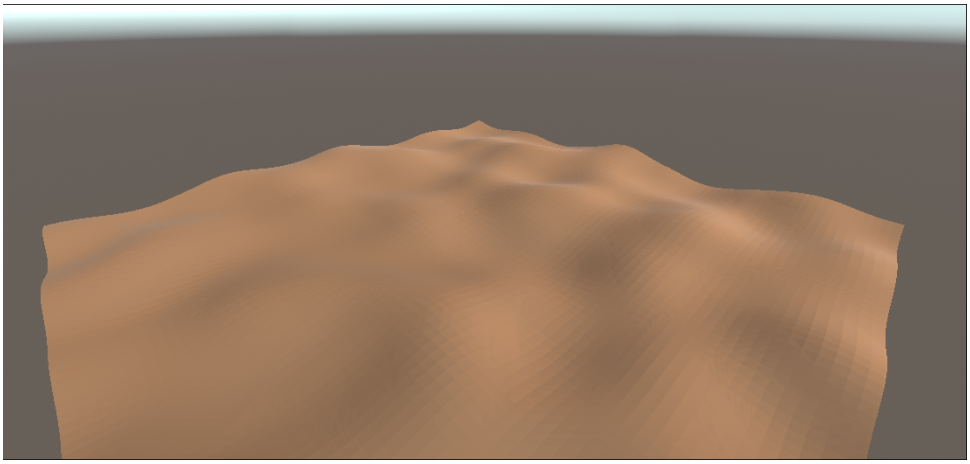


Figure 14: *Density function with horizontal noise of (1,1) and vertical noise with variation 1 and amplitude 4. we can see that the vertical noise dominates the horizontal noise which results in layered terrain.*

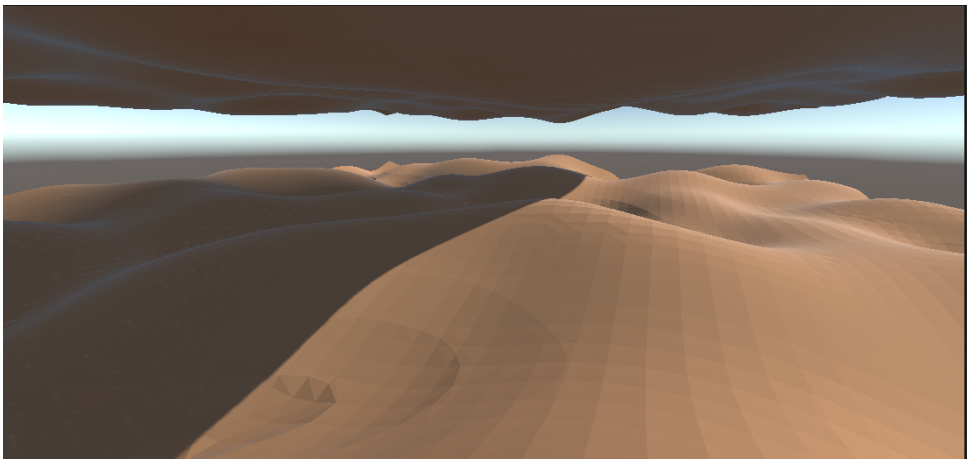


Figure 15: *Layered terrain as viewed from under the top layer.*

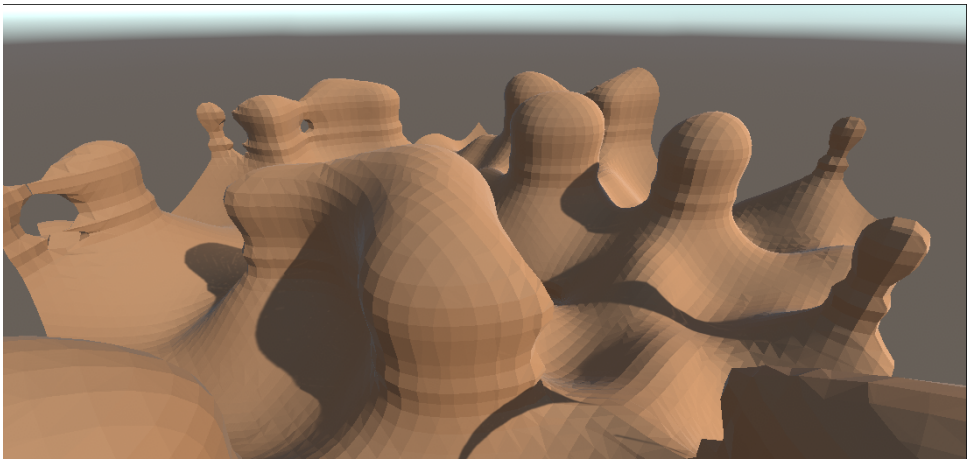


Figure 16: *Density function with both vertical and horizontal noise with a variation of 1 and an amplitude of 4. We can see that adding these two noise layers together generates bulb like structures and some arches can be seen aswell.*

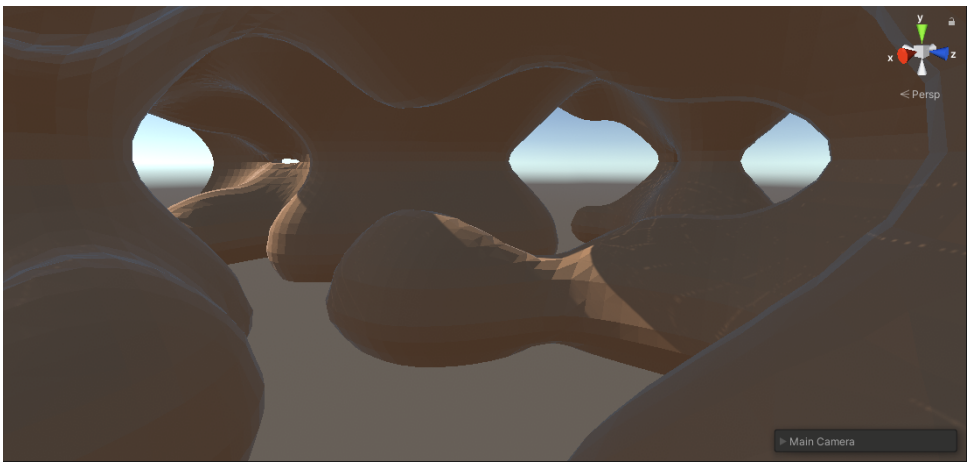


Figure 17: *Underground view of the terrain with a density function of equal horizontal and vertical noise with both having variation set to 1 and amplitude set to 4. We see that large concave structures are formed.*

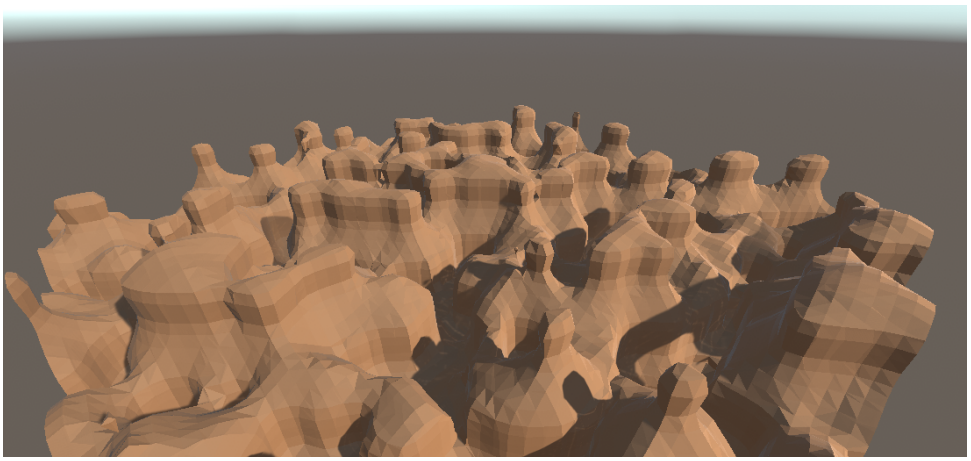


Figure 18: *Above ground view of our final configuration for the density function. We used two layers of horizontal noise (variance, amplitude) (4, 0.125), (2, 0.5) and two layers of vertical noise (1, 0.5), (4, 0.125)*

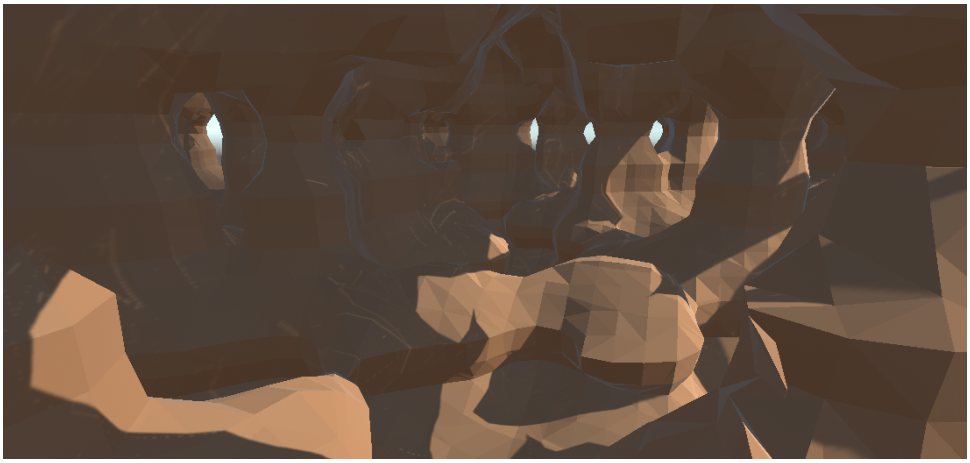


Figure 19: *Underground view of the terrain with a density function with our final configuration we see a variety of edges and banks forming underwater.*