Viktoria Kovecses

260482300

# Simulating an Underwater Environment

## 1. Introduction

The main objective of this project was to create a realistic underwater environment with the use of procedural content generation and steering behaviours. In other words, the goal was to create a game level in which a character is standing at the bottom of the ocean and can walk around to observe and interact with various sea creatures. In addition to this, the ocean floor is continuously created and destroyed as the player walks around, so as to make it appear infinite.

Two different techniques were used to create the terrain using procedural content generation. One was to create a set of tiles with varying quantities and displacements of vegetation and rocks, and then to randomly select and place these tiles next to one another on the game map. Then as the character moves toward the edge of the tile map, tiles are deleted from behind him and newly generated tiles are added in the direction he is headed. The other technique used was to first create the game terrain using the Terrain GameObject type in Unity3D, and then modify the heightmap of the terrain so as to produce small bumps and valleys (i.e. sandy dunes). After modifying the terrain, rocks are randomly distributed across the seascape.

As for the underwater creatures, these were animated based on the use of several different steering behaviours. Depending on the type of sea creature, a different combination of wandering, fleeing, and flocking behaviours were used to control the animal’s movements. Furthermore, to ensure that the fish do not wander too far from the player (so that we can see fish in every simulation), there is a circular perimeter around the player that the fish must try to stay within. This was done by adding a flee behaviour away from the perimeter, to any fish that travels too far.

The results showed that using steering behaviours was sufficient to create interesting and realistic fish movement patterns, as expected. However, the collision avoidance system leaves room for improvements as although collisions were rare for reasonable numbers of fish, they still occurred and made the simulation slightly less realistic. This was because the collisions did not appear natural, as the fish were rigid and bounced around in odd directions when colliding. In addition to this, modifying the heightmap of a terrain GameObject provided much better terrain results, and terrain generation was faster using this method on larger maps, when compared to the tile based method. It created an interesting “sandy dune” effect, which made the terrain look more organic than the flat terrain made using tiles.

## 2. Background

### 2.1 Modelling Fish Behaviours

Previous work done on the modelling of fish shows that the simple steering behaviours described by Reynolds are very powerful in creating realistic fish movements [1]. For instance Pham, Stephens, and Wardhani, combined steering behaviours with simple motivations to model the behaviour of fish and sharks [2]. They based their models on observations of real fish movements, their behaviours (i.e. predator evasion, group formation, and hunting), as well as three senses (mechanoreception, vision, and chemoreception), in order to achieve the most realistic models. In addition to using different combinations of the basic steering behaviours of wandering, flocking, seeking, and fleeing to move the fish, they also introduced motivations. For instance, if a shark hasn't eaten in a while it would become hungry and more motivated to seek out prey. Here, similar to this research, steering forces were applied to a point mass on each fish (i.e. the center point of the rigidbody attached to the fish) to move it around in the environment. However, motivation was removed from the model as the main interest was to see if realistic fish movement patterns could be created based on steering behaviours alone.

Another experiment with fish behaviour done by Huth and Wissel focussed specifically on modelling the schooling behaviour of fish [3]. They made a few basic assumptions about fish, two of the main ones being that a fish is only influenced by its nearest neighbours, and that the group swims without a leader. After performing various experiments to find the best way of imitating real fish, they found that fish schools can be realistically modelled with the use of the three steering behaviours used for flocking described by Reynolds, namely separation, cohesion, and alignment [1]. In addition to this, averaging the influences of the neighbours produced better results than if each fish chose its movements based on only one of its neighbours [3].

### 2.2 Terrain Generation

Game terrains are often generated procedurally by creating random heightmaps using techniques such as spectral synthesis (similar to Perlin noise), and midpoint displacement strategies, such as the diamond-square algorithm described later [4]. However, the problem with these methods is that they usually create coarse heightmaps representing rugged terrain. Simulating erosion can aid in smoothing the terrain, either through thermal (material from high areas breaking off and falling to lower areas), or hydraulic erosion (based on rainfall and water movement) [5]. While these erosion algorithms may be useful, they are also very slow and not efficient for a game where content is continually being generated. Therefore, when creating the terrain for this project, in order to ensure smooth sand dunes, the heightmap of the terrain was decreased so that each point of the heightmap controlled a larger area of terrain, which would allow better smoothing to be performed implicitly by Unity.

One technique that has been used for generating terrains that appear infinite is the use of adaptive tiling to only generate the terrain visible from the player's current position [6]. In order to ensure consistent terrain, the tiles are stored in files so that when a player revisits an area, the same tile can be regenerated. As the player moves around, terrain is loaded in the direction the player is headed, and terrain is placed around the player so as to ensure a 360 degree view. In addition to this, tiles are removed as the player moves away from an area. A simpler technique is implied in the 2D game generated by Johnson, Togelius, and Yannakakis, who create infinite cave levels [7]. Here they generate a central tile, and then generate a tile next to each edge of the central tile so the player can move North, South, East, and West. Here, a similar technique to these was used, with the tiles newly randomly generated as they are placed, as it was considered to be unreasonable to expend memory storing the tiles when there are not any clear discernable landmarks that the player would notice anyway.

## 3. Methodology

Two different methods of procedural terrain generation were experimented with in order to create the ocean floor in the simulation/game level. The first was to manually create tiles representing small portions of the ocean floor, by placing a few different rocks on each tile. A grid of a predetermined size would then be created by randomly selecting and placing these base tiles. The tiles would then be deleted and created as the player moves around to create a sense of an infinite world. The other method used for terrain generation was to create larger tiles at runtime by modifying the heightmap of the Terrain GameObject in Unity and placing 9 of these in a 3 x 3 grid. As for the fish, they were split up into three types namely wandering fish, schooling/flocking fish, and crabs. The wandering fish made use of the wandering steering behaviours, while the schooling fish mainly relied on flocking behaviors, and the crabs made use of leader-following techniques. In the absence of other fish of the same species, schooling fish and crabs also relied on wandering behaviours to move around the game world.

### 3.1 Procedural Content Generation

#### 3.1.1 Generating Infinite Terrain with Tiles

The tiles used for constructing the grid were generated manually using a Plane GameObject of fixed size. No more than three rocks were placed on each tile, in such a manner so as to mimic how they might be distributed on the sea floor in real life. Five different tiles proved sufficient to create interesting terrains. An algorithm then randomly selected the tiles and placed them into a grid, which was maintained using an n x n array. On selection, each tile was also assigned a random rotation of 0, 90, 180, or 270 degrees, which helped to create variable and interesting terrain with only a few tiles.

In order to make the terrain appaear infinite, the player's position was always maintained and tested to see if he was still standing on the tile at the center of the grid. If he were to move in any direction, and step off of the central tile, then either the last row or column (or both) of tiles behind him are removed and a new row or column of randomly generated tiles are placed at the last row/column directly in front of the player (see Figure 1).

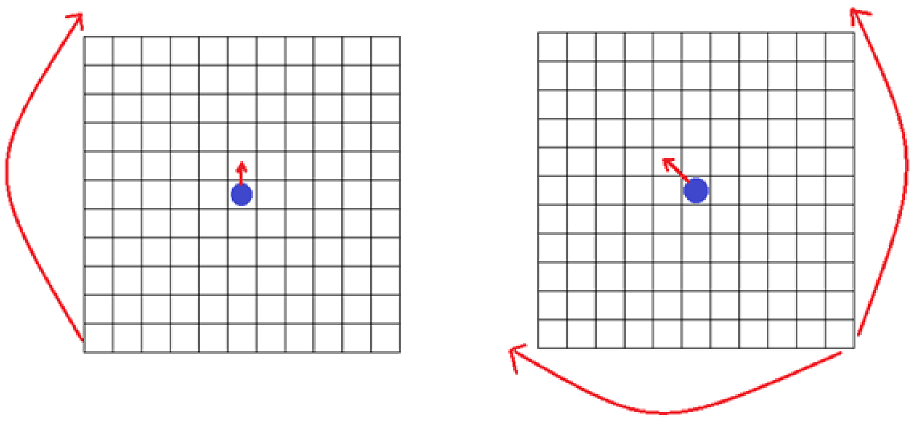

*Figure 1:*
***On Left:*** *If the player moves upwards, then the last row of the grid behind him is moved to the last row in the direction the player is headed. The grid cells are actually newly generated as they are moved.*
***On Right:*** *If the player moves diagonally, then both a column and row of grid cells are moved in the direction he is deaded.*

Although the positions of the tiles in the game world are physically moved so as to ensure that the player is always in the center of the grid, the location of the tiles in the array remains unchanged. Changing the positions of the tiles in the array would require an expensive shift of every array cell to a new position, which would have been inefficient and costly. Therefore, the tile positions do not change, however two pointers are maintained to indicate in which row/column the bottom and right edges of the grid are currently situated. For instance, if the player moves upwards on the grid, then the bottom cells are moved to the top of the grid, and the pointer to the bottom of the grid shifts upwards by one (see Figure 2).

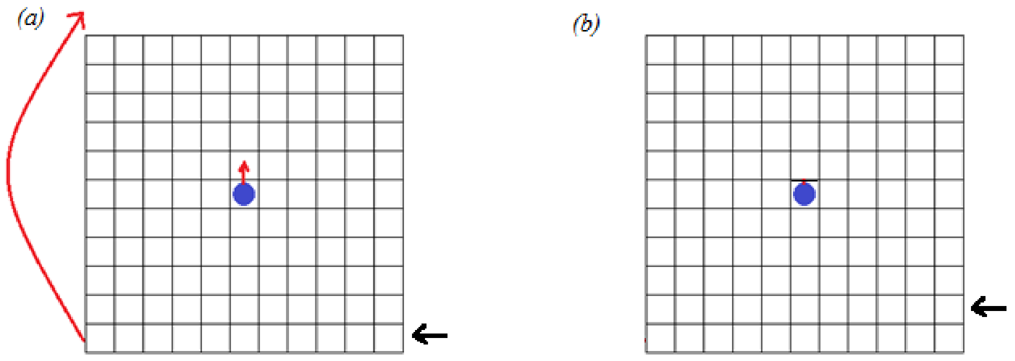

*Figure 2:*
*The small black arrows point to the bottom row of tiles in the array before (a) and after (b) the player moves.*

### 3.1.2 Generating Terrain with Diamond-Square Algorithm

The other method for generating terrain used the diamond-square algorithm to modify the heightmap of a terrain GameObject. The diamond-square algorithm is a type of midpoint-bisection algorithm which generates heights for a square grid with the four corners already initialized. It then computes the average of these four corner values and adds/subtracts a bit of noise to find the value for the center point of the grid (this step is known as the “diamond step”). This is followed by the “square step”, which involves computing the center of each edge by averaging the two corners that lie on that edge, and once again adding/subtracting random noise. The algorithm then splits the square into four smaller squares, and runs the algorithm recursively one each square until a height is found for each point [4] (see Figure 3). This algorithm was run twice on the heightmap, once to create bumpy terrain, and the next to smooth the terrain by a small degree.

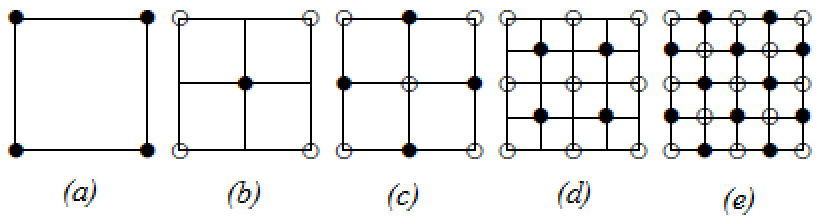

*Figure 3:*
*The first five steps of the diamond-square algorithm. (a) Shows the four corner points initialized, (b) Shows the diamond step where the corner points are averged to find the center point, (c) Shows the square step where the center of each edge is found by averaging the two corners that lie on that edge, and (d) and (e) show the process continueing recursively.*

The corner and edge points of the heightmap were then converted back to base level, so that other terrains could easily be joined to this terrain at the sides. Next, rocks were randomly added to the terrain using a uniform distribution. Their heights were randomly chosen based on the heights of the terrain such that only part of the rock would be visible, and the rest buried in the sand. A total of nine terrains were created this way and placed into a 3 x 3 grid to create the game world. Based on this grid we could then do the same infinite content generation that was done for the tiled grid.

## 3.2 Steering Behaviours

### 3.2.1 Wandering Fish

The first type of fish to be created was the wandering fish, which relies mainly on the “wander” steering behaviour described by Reynolds to move around [1]. Each fish had a sphere placed in front of it, and chose the direction and force of its movement by picking a random point on the sphere and moving towards it. If the fish picked a point closer to itself, then the movement was slower than if it picked a point which was further. In addition to this wandering behaviour, the fish also used a simple obstacle avoidance strategy which involved projecting its position into the current position of the sphere and then projecting slightly further to see if it would collide with any objects if it continues moving in this direction. If there is a collision, then the fish calculates the force and direction it would have it if had collided with the object (based on the normal force). This vector is then added to the wandering force. However, before applying this movement force to the velocity of the fish, there is first a test to see whether this force would cause the fish to move in an unnatural way (i.e. move up or down at an angle greater than 50, or even move upside down), and scaled down if it does. The force is then used to change the velocity and direction of the fish. The velocity is then normalized and multiplied by the maximum speed to keep the fish from moving too rapidly.

### 3.2.2 Schooling Fish

The next type of fish to be created was schooling fish, which form groups when in close enough proximity to other fish of the same type. The main steering behaviours used to direct these fish were separation, cohesion, and alignment [1], with most importance given to cohesion and alignment. Separation was used to keep the fish from colliding with one another as they swam, and was calculated by summing the vectors that would move the fish away from each of its neighbors. The cohesion force, which kept the fish together, was obtained by averaging the positions of each of the neighbors, and adding a seek force towards that point. As for the alignment force, which ensured that all the fish swam in the same direction, it was calculated by finding the average direction the neighbors were headed and adding a force in this direction to the other forces (see Figure 4 for summary of these forces). If the fish could not see any other fish of the same type within a 240 degree angle in front of it, it would use wandering behaviour to move around until it could join a group. In addition to these forces, this type of fish also used the simple obstacle avoidance strategy described for the wandering fish, and had its upwards/downwards movements constrained in the same way as well.

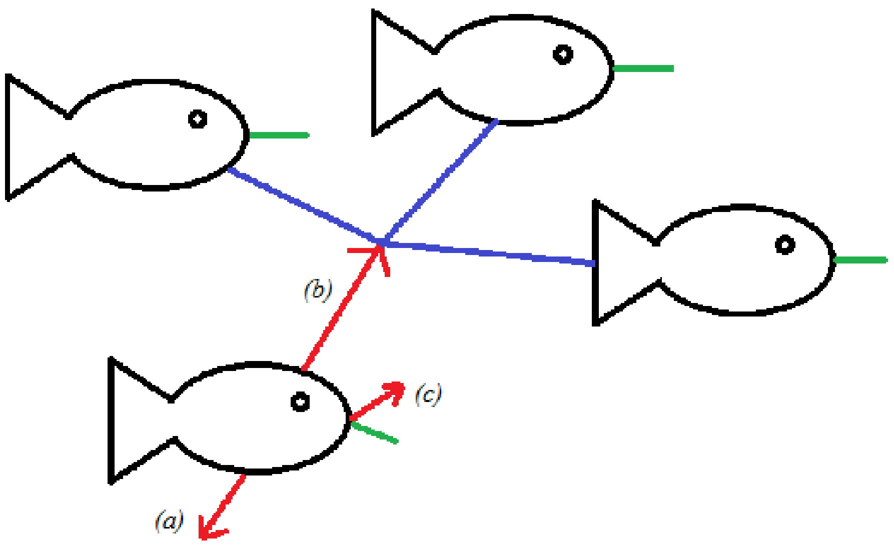

*Figure 4:*
*(a) shows the seperation force acting on the fish*
*(b) shows the cohesion force acting on the fish*
*(c) shows the alginment force acting on the fish*

### 3.2.3 Crabs

The third and final type of underwater sea creature created was the crab. Crabs were limited to movement along the ocean floor, and displayed a leader-following steering behaviour when behind another crab. Before the crab finds a leader for itself, it uses a wandering behaviour using a circle situated in front of the crab to choose its direction. However, once the crab has found a leader it follows the leader for the rest of its life span in the game. In order to implement the leader following behaviour, target arrival and separation steering behaviours were used [1]. The target arrival behaviour was done by having the crab seek towards a point slightly offset behind its leader, and reducing the seek force as it got closer to this point. The separation behaviour was implemented in the same way as for the schooling fish, to prevent crabs from bumping into each other if they were following the same leader. In addition to this, the crabs were affected by gravity so as to stay close to the ground; however, if they got too close to the ground they began to tip over, therefore there a height was specified at which the crabs should remain. If the crabs go below this height the force of gravity was countered to keep them from falling down. Once again, crabs also use the simple collision avoidance policy described for the wandering fish to prevent them from bumping into rocks scattered around the ocean floor.

### 3.2.4 Perimeter Constraint

Finally, one last steering behaviour was added to all of the sea creatures to keep them from travelling too far away from the player. This was added to help ensure that there were always at least some fish visible for the player to observe. The fish were allowed to move around freely within a dome around the player, however, as soon as a fish left this dome, it would immediately begin to flee back towards the player. The direction and force vector for this was calculated based on the distance and location of the fish relative to the player, so that the further the fish was, the greater its fleeing force.

## 4. Results

The goal of this project was to simulate an underwater environment that could be created at random procedurally. In order to test whether this was successful, it was important to analyze the visual components, as well as how fast content could be generated. There was no way to quantify how well the movements of the simulated fish resembled actual fish movements, however just looking at a screenshot of the scene gives a good indication of whether this looked like a realistic underwater environment (Figure 5). The fish are all swimming at realistic angles and directions, with the schooling fish (blue) swimming together. Therefore, the simulation does indeed

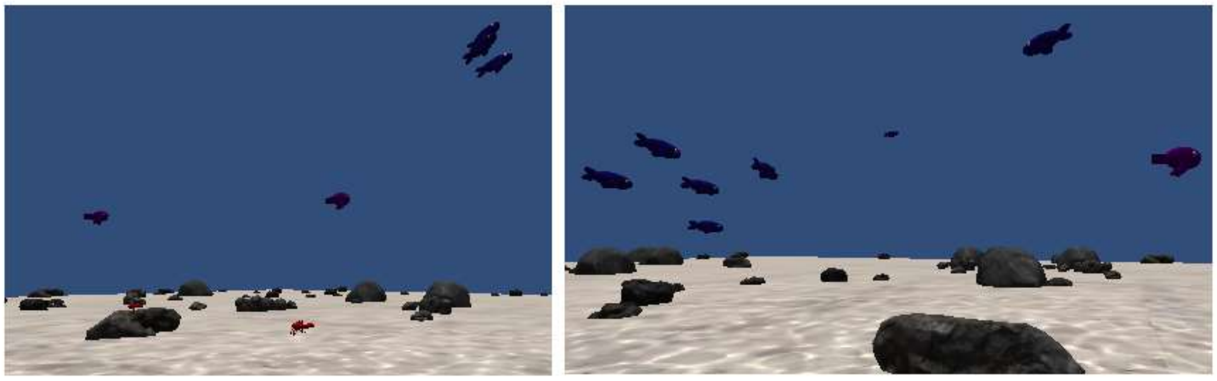

*Figure 5:*
*Two screenshots of the underwater scene taken about a minute after start of simulation. Can see schooling fish (blue), wandering fish (purple), and crabs (on left panel).*

resemble an underwater environment containing various kinds of fish. The number of fish necessary to produce a decent underwater environment using these techniques was also determined by observing the number of fish visible in a fixed camera per frame (1000 frames were averaged for this), as well as the number of collisions that occur, depending on the number of fish in the game (Figures 6 and 7).

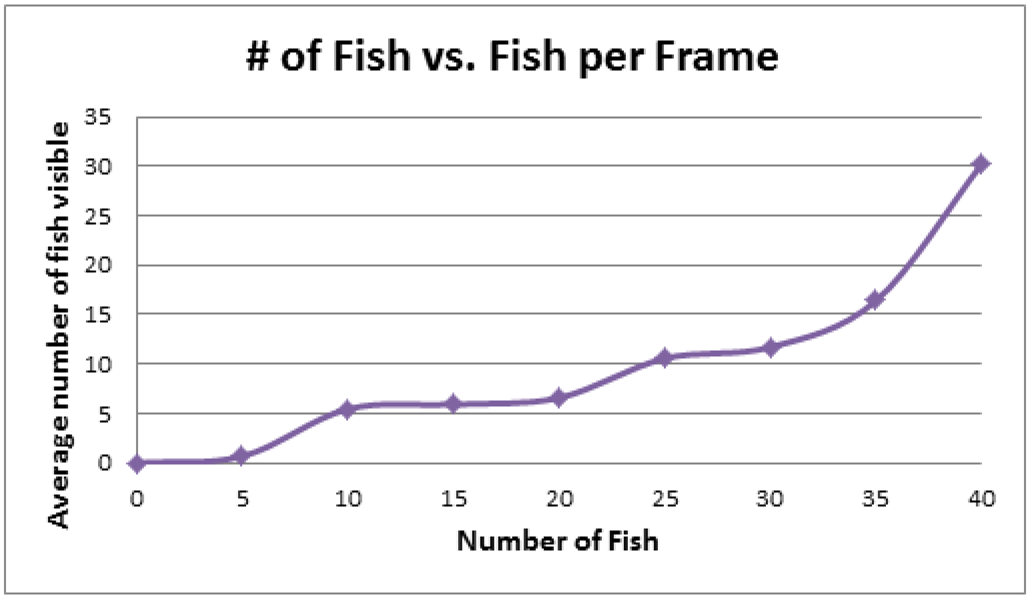

*Figure 6*

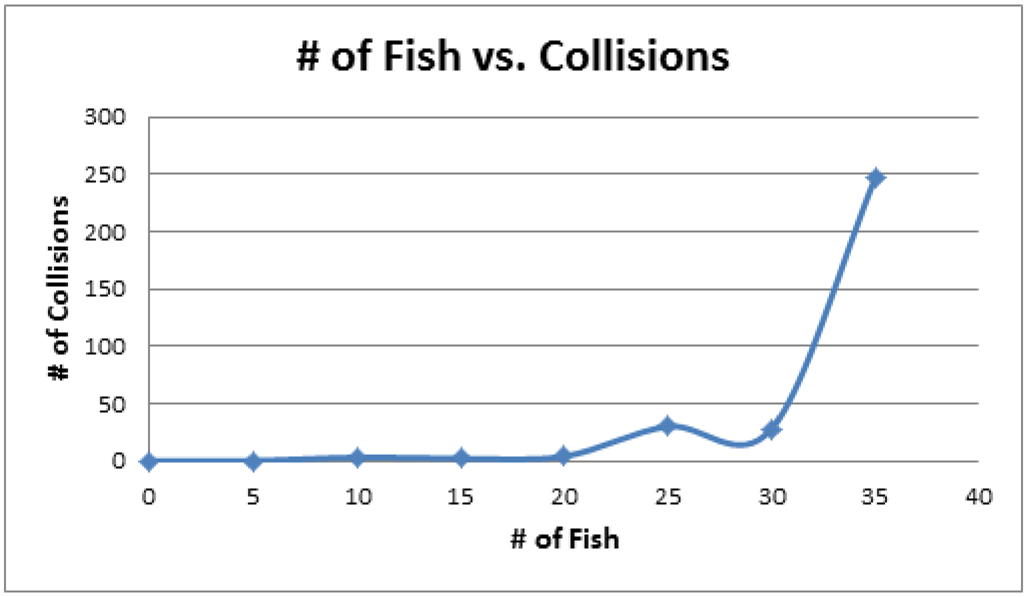

*Figure 7*

Based on these two graphs, it was determined that having between 10 – 25 fish in any given simulation would be both necessary and sufficient. While having less than 10 fish would be too few as the player would not be seeing enough at a time for the game to be interesting. Although, having more than 25 fish may significantly increase the number of fish visible per frame, and thereby make the game level more interesting, this would even more dramatically increase the number of collisions occurring. With so many collisions the game begins to become unrealistic as fish do not often crash into one another naturally. In addition to this, although not measured precisely, the game begins to noticeably slow down for more than 25 fish, and becomes dramatically slow and choppy at 40 fish. Therefore, it would not be reasonable to generate more than about 20 fish, especially since this is only supposed to be the background for the level and does not take into account the computations necessary for the actual gameplay that one may add to make the game more interesting.

As for the terrains, when comparing the two different seascapes generated (tile based vs. terrain based), it was found that the tile based approach was inferior based on a couple of crucial observations. Firstly, the tiles used were flat and therefore did not provide any interesting dimension to the ground (e.g. sand dunes or hills), but simply a flat and uninteresting landscape sprinkled with rocks. On the other hand, the terrain based approach included interesting bumps and ridges along the ground, creating an uneven surface much closer to being a believable model of the bottom of the ocean floor (although it is plausible that the ocean floor contains many flat regions as well) (see Figure 8). In addition to this, the amount of time required to render the seascape was also taken into account (Figure 9). It was predicted that the tile based approach would render faster, as it requires nothing more than having tiles randomly selected and added into a grid, whereas the other method also required the production of a heightmap as well as the addition of 50 randomly placed rocks along the terrain surface. And indeed for seascapes that are less than approximately 150 world units this is true. However, for larger seascapes the tile based approach appears to increase linearly, whereas the terrain based approach takes almost the same amount of time to render regardless of the size of the terrain.

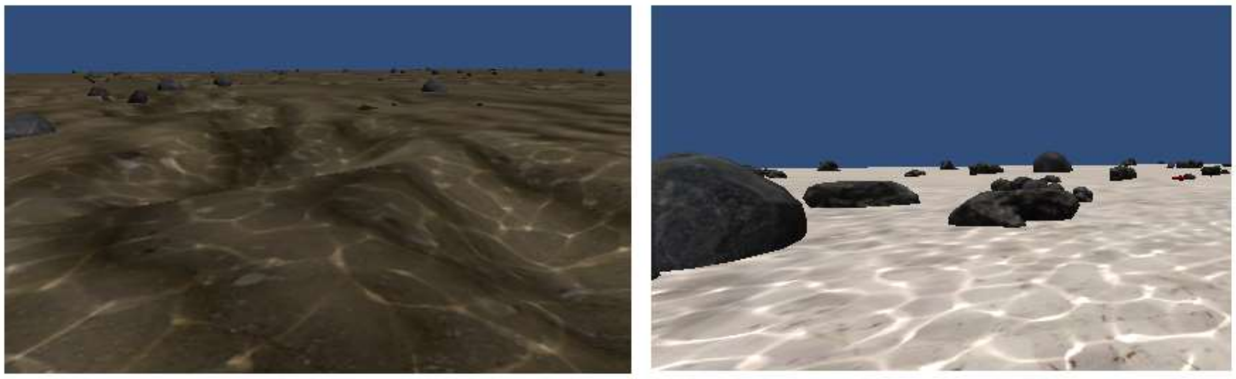

*Figure 8:*
*Left: Seascape generated by changing the heightmap of the terrain GameObject (terrain based method)*
*Right: Seascape generated using tiles chosen at random (tile based method)*

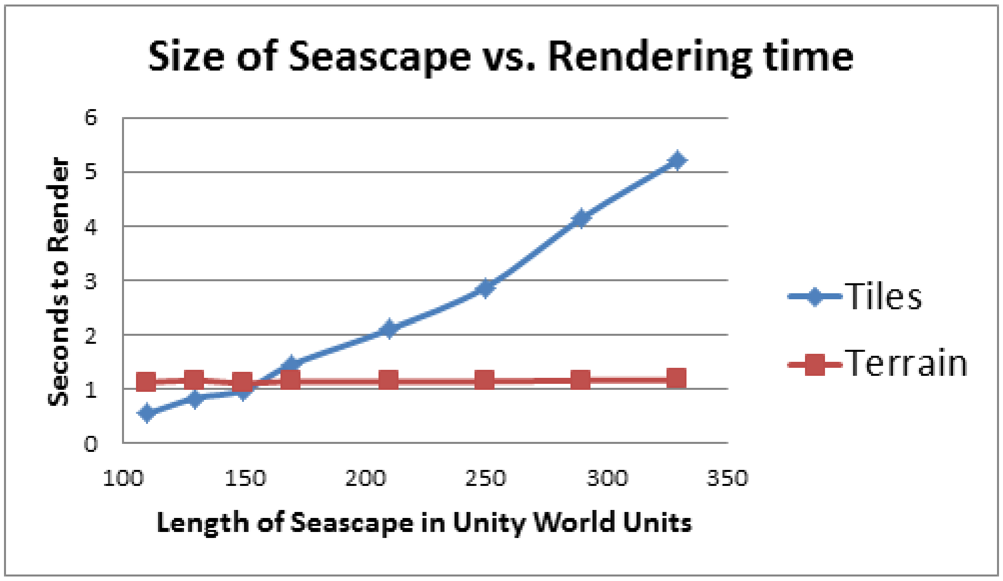

*Figure 9*

Therefore, based on these observations, it was concluded that a decent seascape could be created using the methods tested, as long as there are less than 25 fish in the game, and with a preference towards the use of the terrain GameObject to procedurally generate the terrain.

## 5. Conclusion

In summary, the project was overall successful in creating a realistic underwater environment, although there are still many improvements that can be made to improve existing functions and add more interesting features. The main focus was to not necessarily give the marine animals and landscape the most accurate representation of how they would behave/look in real life, but to give the scene the appearance of being underwater. In other words, to create something representative of how most people imagine the bottom of the ocean, and make it “realistic” in this sense. It would have been interesting to explore the creation of specific fish species and have them move based on how their real life counterparts behave. For example, the crabs do not necessarily behave as real crabs, although they may appear to. Some crabs actually display more of a flocking behaviour, especially when threatened, such as the fiddler crab [8]. However, instead of implementing flocking behaviour, leader-following was implemented so that a larger variety of steering behaviours could be observed and experimented with. In addition to this, an improved collision detection system could produce better results as even a few collisions between fish may be unrealistic. In addition to this, a larger variety of fish could be added to create much more interesting scenarios. For example, the addition of a shark which seeks and eats any fish that it is near enough to, while the fish attempt to flee from the sharks. Furthermore, other methods for generating the heightmap of the terrain could be explored to compare with the diamond-square method, or other features added to the terrain such as coral reefs. Finally, it would be interesting to have a complete game produced with the use of this simulation as a background for the game level. Especially as there is no way to tell how much the simulation may slow down with added features, and what would be a reasonable amount of fish given this increase in rendering time.

## References

[1] C.W. Reynolds, "Steering behaviors for autonomous characters," In Game developers conference, pp. 763-782, 1999.

[2] B. Pham, K. Stephens, and A. Wardhani, "Modelling Fish Behaviour," Proceedings of the 1st international conference on Computer graphics and interactive techniques in Aurstalasia and South East Asia, ACM, 2003.

[3] A. Huth and C. Wissel, "The Simulation of the Movement of Fish Schools," Journal of theoretical biology, vol. 3, no. 156, pp. 365-385, 1992.

[4] J. Olsen, "Realtime procedural terrain generation-realtime synthesis of eroded fractal terrain for use in computer games,", 2004.

[5] S. A. Groenewegen, K.J. Kraker, and R. M. Smelik, "A Survey of Procedural Methods for Terrain Modelling," In Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS), pp. 25 - 34, 2009.

[6] J.E. Marvie and J. Pouderoux, " Adaptive Streaming and Rendering of Large Terrains using Strip Masks," In Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, ACM, pp. 299-306, 2005.

[7] L. Johnson, J. Togelius, and G.N. Yannakakis, " Cellular automata for real-time generation of infinite cave levels," In Proceedings of the 2010 Workshop on Procedural Content Generation in Games, ACM, pp. 10, 2010.

[8] S.V. Viscido and D. S. Wethey, " Quantitative analysis of fiddler crab flock movement: evidence for 'selfish herd' behaviour," Animal Behaviour, vol. 63, pp. 735-741, 2002.