# Procedural Generation and AI Learning in an RPG game

**Alan Quigley**

**BSc. Software Design 2016**

**Athlone Institute of Technology**

**Declaration**

I hereby certify that the material, which is submitted in this report towards the award of BSc. Software Design, is entirely my own work and has not been submitted for any academic assessment other than part fulfilment of the above named award.

Future students may use the material contained in this report provided that the source is acknowledged in full.

Signed…….

Date……

**Abstract**
Procedural generation of content in games has recently become a huge area within the gaming industry particularly on the indie development side of the gaming industry. Procedural Generation allows a games longevity to increase by adding a massive amount of replay ability to any game, with some companies creating an entire galaxy using procedural generation, where no planet is ever similar. A.I. learning is currently very underutilised in gaming, with many game A.I.'s just interacting with the player but never reacting to the player. This project aims to combine these two things, procedural generation and A.I. learning, into one system which can generate enemies and also learn from how the player is playing in order to choose the best suited enemy.

**Acknowledgements**
I would like to thank Dr.Mark Daly for his continuous guidance and assistance with this project. I would also like to thank John Barrett for his feedback on the development of this project.

I'd like to thank my family for their support during the project.

I'd like to thank Gareth Kennedy, Shane Kelly and Calum O' Donoghue for supplying me with "Top quality banter" over the course of the project.

**Table of Contents**

**List of Tables**



**List of Figures**

# Chapter 1: Introduction

## 1.1 Introduction

Procedural generation is becoming a larger part of the gaming industry, and often used as a mean of prolonging the longevity of a game through replay ability, while artificial neural networks are still relatively unused and rarely considered to be used in order to tackle problems in games. This project aims to combine these two ideas, neural networks and procedural generation, in order to make generating thousands of, in the case of this project, enemies while also showing that neural networks can be used to great effect in games. This report shall cover the research undertaken in this project, the design of this system and its implementation in both a high level and low level design as well as testing of the system and the final conclusions gathered from the project.

## 1.2 Context and Rationale

The context of this project is to create a system that can procedurally generate content while also contains some form or another of A.I. learning. This system should be able to be applied to any RPG style game that wishes to procedurally generate its enemies instead of hardcoding the enemy stats. A Self-organising map will be used to create this system. The reason the SOM was chosen, which is explained in more detail in the following chapter, is because it allows for thousands of enemies to be generated within it, thus solving the procedural generation problem, while also organising them through use of an unsupervised learning technique.

## 1.3 Research Aims and Objectives

The research aims of this project is to gain an understanding of SOM and the intricacies of how they function, and if they would be suitable for use in procedural generation and A.I. learning. The objectives of this project are as follows:

- To create a fully functioning SOM in unity.
- To use this SOM to generate thousands of possible enemy stat combinations.
- To have the SOM be able to get the node with the best fitting nodes for a scenario.
- Finally, to have a system that does procedurally generate data and contains A.I. learning.

# Chapter 2: Background Research

## 2.1 Introduction

Artificial Neural Networks have the potential to achieve great things in terms of gaming. The following are a couple of ways ANN's can be used in gaming.

### *2.1.1 Uses for Artificial Neural Network's in Games*

ANN's can have many different uses in games for controller and creating a realistic AI. Currently the field of robotics are using neural networks to control robots and have them move in a realistic fashion. In the case of the robot, its sensory system will send relevant inputs to the neural controller which in turn sends the relevant response, which may consist of several output nodes, to the robots' motor control system. An example of this would be if the robot was detecting if it had an obstacle in front or to either side of it, the distance of the sensed obstacle may also be taken as an input. The neural controller of the robot will have outputs to control the movement of its legs with the resulting output send by the controller making the robot walk forward, turn left or right, or possibly walk backwards. This type of control would be perfect for use in games to move control NPC's be they tanks, spaceships, airplanes or even biological characters, humans, dogs etc. Some driving games have used neural networks to control the AI Cars in a race, with the most notable example being Colin McRae Rally 2.0 which was developed and published by Codemasters. In Colin McRae Rally 2.0 the neural network used to control the AI cars was trained in a learning by example method where the network was trained by observing how the game developers themselves raced around the courses in the game. [1]

ANN'S can be used in strategy games as a way for the computer controlled enemy to predict the level of threat the player poses. Let's say this strategy game requires the players to build technology and train units to attack or defend from the computer controlled enemy. The AI could try and predict the level of threat posed by the player by taking in a number of inputs such as the players; ground units, aerial units, if the ground units are moving, if the aerial units are moving, the range of the ground units and finally the range of the aerial units. These inputs can result in four possible threat levels for the AI, that the players ground units are attacking, their aerial units are attacking, both ground and aerial units are attacking or that the player is not attacking. Based on these possibilities the AI may behave differently from constructing more defences in the event of a full scale attack to carrying on as normal if there is no threat

currently. This type of network would require in-game training and validation but has the potential to calibrate itself to the players play style. This also reduces the workload of having to figure out every possible scenario and threshold which can be encountered if using a rule based finite state machine AI. [1]

### *2.1.2 Introduction to Neural Networks*

The first ever neural network was created in 1956 by Frank Rosenblatt. This first neural network was called ‘Perceptron’ and thirteen years after its creation, a publication by Papert and Minsky called ‘perceptrons’ was released. This publication would formalise the concept of neural networks, but also point out the huge limitations of this original architecture. It revealed how perceptron was unable to perform a basic operation of an exclusive-or. [2] This revelation almost stopped research into neural networks, however it did not and since then neural networks have become a huge area of research and are massively popular. [3]

Neural networks are designed to mimic the modelling of the brain. Unlike standard computer science concepts, which typically store data in a frame which is then stored in a centralised database, neural networks store their information throughout the network. This method of distributing information is based on how the brain stores information in memories which are stored in synapses in the brain. [3]

Part of the beauty and appeal of neural networks is their ability to be extremely fast and efficient. This speed and efficiency allows them to handle huge piles of data. The reason they can do this all boils down to the fact that each node in the network is its own autonomous entity. The nodes only perform a small computation relative to the size of the problem. The sum of all these nodes is where the true potential of the neural network resides. AS a result of this architecture parallel implementation is present, similar to how the brain functions which does nearly all of its actions in parallel. [2]

A great benefit of neural networks is that they are fault tolerant. A bunch of non-functioning nodes, or some bad data is not going to cripple the entire system. However, this tolerance is only to a point. Too much bad data will produce noticeably incorrect results. Which again is similar to the human brain, where a slight injury to the head may not result in any difficulties, but a large injury or a slight injury to a certain point of the head will. [3]

### *2.1.3 How do neural networks learn?*

In terms of biology, learning is an experience that changes the state of a living organism such that this new state improves how the organism performs in a subsequent similar situation. In terms of the neural network, it must learn of the given information and then, after it is trained, it can then be used for pragmatic purposes. [3] This idea of learning can be divided into two subsections, Supervised and Unsupervised learning.

Supervised learning uses an external teacher to show the network what the desired outputs are to a give input signal. Global information may be required during this type of learning. [4] This way of learning utilises both input and output vectors, with the input vectors being used to provide the starting data that output vectors compare with to determine some degree of error. Reinforcement learning is a type of supervised learning where the network is only told if it is right or wrong. [3]

Unsupervised Learning requires no external teacher and only uses local information. This style of learning can also be called self-organisation as the network self-organises its data and detects its emerging collective properties. [4] In other words the network does not know what the correct answers are and must figure out the patterns in the data by itself. This type of learning only uses input vectors and does not use the output vectors to learn from. The most important thing about this type of learning is that it does not require any human interaction, which can be beneficial when dealing with large amounts of data that would be hard for a human to compute. [3]

As you can see there can be many different types of neural networks. The neural network chosen for this project is the Self Organising Map which will be discussed next.

### *2.1.4 My Rationale for using an SOM*

An SOM, was chosen for this project as it classifies, clusters and organises high dimensional data into a lower dimensional form while also preserving the topological relations of the input data. This allows for the selection of several nodes that are ‘close’ together. This is beneficial for when the chosen node fails to defeat the player, as one of the other nodes with similar weights can be chosen to battle the player. This new node should be ‘further’ from the player than the original chosen node, further in the sense that it will be stronger, having a larger weight in the desired attribute. For

example, if the player has a high strength skill of seven and an endurance stat of six, the best matching node will have similar if not identical strength and endurance stat, however if the BMN loses then the next chosen BMN should be stronger in either strength or endurance without compromising the other stat by having it be lower. This process of selection will repeat until the player is defeated by an enemy.

## 2.2 Self-Organising Maps

### *2.2.1 Introduction to SOM's*

A Self-Organising Map is an ordered mapping of a set of given input data onto a usually two dimensional grid. Each node in the map has an associated model, mi (figure 2.2.1). Every model is computed by the SOM algorithm. input data gets mapped to the node which has the closest model to that of the data item, e.g., if the model has a small Euclidean distance to the input data. Every model is at first usually a certain mean of the given input data in the data space, yet after being computed by the SOM algorithm they become more similar to adjacent nodes than to nodes that are further away from them on the grid. The SOM uses a method of unsupervised learning to achieve its organisation. [5]

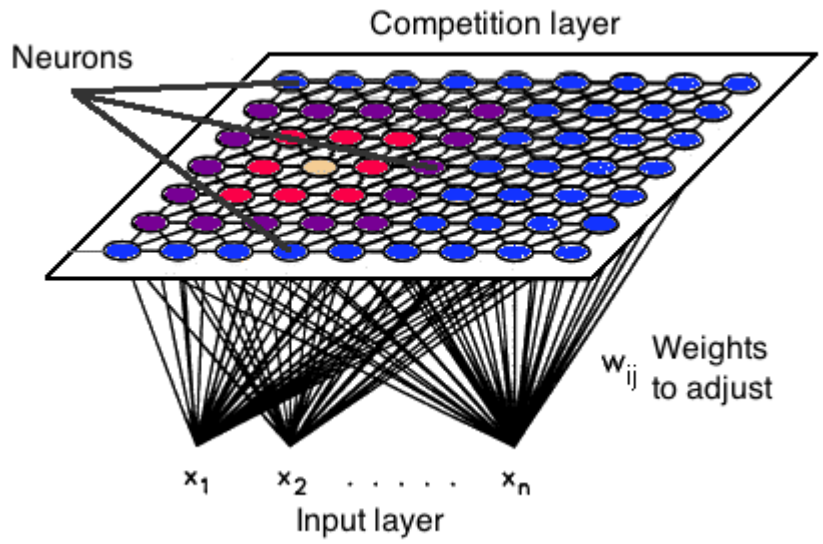


Figure 2.2.1: Visual representation of a SOM

### *2.2.2 History of SOM's*

The Self Organising Map, sometimes referred to as the Kohonen Network, was introduced in the 1980's by Finnish Professor Tuevo Kohonen, as a way of visualising high dimensional data into a lower dimensional space. This SOM algorithm evolved

from the initial neural network models, particularly the associative memory and adaptive learning models. The idea of SOM's and similar neural networks was to explain the spatial organisation of the brain, namely the cerebral cortex area of the brain. The first attempts at this were the spatially ordered line detectors by Von der Malsburg and the neural field model of Amari. Unfortunately, since being some of the first to attempt this, these initial neural networks were fairly weak when it came to self-organisation. Kohonen invented a system model which was made up of at least two unique interacting subsystems, which proved to be crucial. The first of these subsystems is a competitive neural network that contains a winner-take-all method, the second subsystem is controlled by the neural network and modifies the synaptic plasticity of the neurons in learning, meaning it will either by increasing and/or decreasing the weights of the neuron. Because of this separation of the two subsystems, a robust and very effective self-organising system can be implemented. [5]

### *2.2.3 SOM Algorithm*

The process in which SOM's go about organising is through the competition to represent samples. The nodes are allowed to change themselves through learning so that they can become closer to the input samples and with the chance of being victorious in the next competition/iteration. This competition in learning is what organises the nodes into a map which displays similarities. [6]

The steps involved in the SOM algorithm are as follows: Initialisation, Sampling, Matching, Updating and finally Continuation. [7]

**Initialisation**, this is the first step in the algorithm. In this step the nodes are created and they have their weights set to random values between 0 and 255.

**Sampling**, this is when a randomly chosen training input is taken from the training set and presented to the SOM.

**Matching**, this step involves searching through each node in the network and checking which one has the closest weight vector (in Euclidean distance) to the training input vector, the BMN.

**Updating**, this step checks which nodes are situated in the BMNs' neighbourhood radius, this radius reduces in size over time. The nodes present in the radius adjust their weights to be closer to those of the BMN. The amount the weights are adjusted are based on how close they are to the BMN, the closer they are the more they are adjusted, the learning rate and time constant of the map.

**Continuation**, this step does not really affect the system as such but it does check if the iterations of learning are finished, if not then it returns to the Sampling step and repeats the algorithm from that point until training is complete. [7]

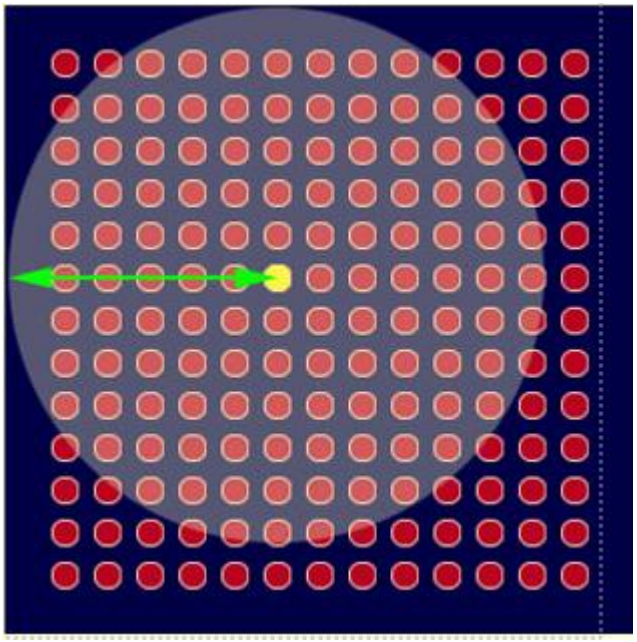

Figure 2.2.2: The neighbourhood radius of the BMN

### *2.2.4 Applications for SOM's*

SOM's are used quite extensively in various different fields, with even an international workshop dedicated to SOM's. Applications that use SOM's range from Vector Quantisation and image compression all the way to density modelling and data visualisation.

#### *2.2.4.1 Vector Quantisation and Image Compression*

Upon the shrinking of the neighbourhood radius to house just the winner, the SOM satisfies the two conditions necessary for VQ. This neighbourhood function provides the SOM with two advantages over other VQ's, the first being that the SOM is superior at conquering the over or underutilisation and local minima problem, the second being

that the SOM produces a map of code vectors with some ordering allowing the map to deal with noise contained in the input or retrieval patterns.

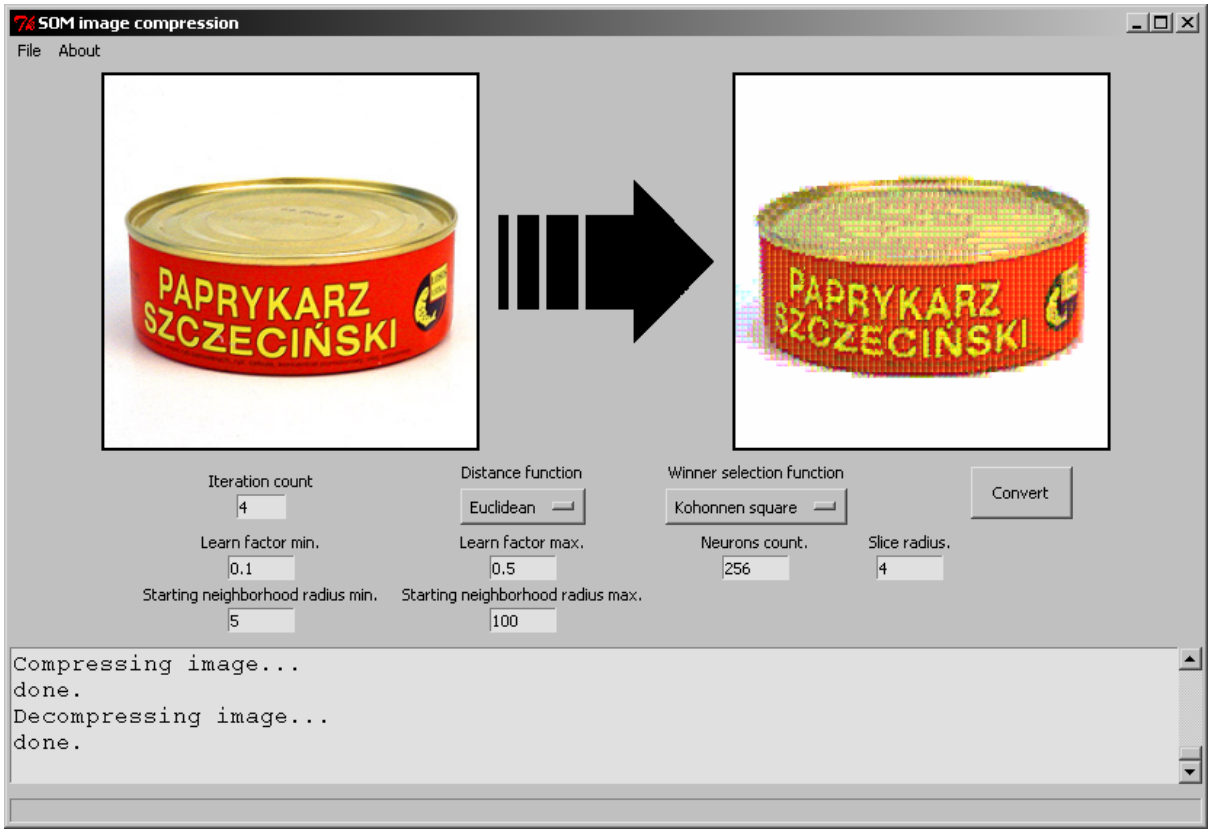


Figure 2.2.3: Program that uses an SOM to compress images, created by lucas Roguski

SOM's tend to outperform other VQ's, particularly when local optima are present. The robustness of the SOM has been improved to the point where the quantisation has been reduced in these improved SOM's, with SOM's also being used in video compression to improve the performance with lower bit rates. [8]

*2.2.4.2 Data Visualisation*

Visualising data is an important area for ANN's, the SOM in particular, is very beneficial to use in this area as, unlike most other neural models, the SOM has a topology preserving property. Identifying clustering tendency, revealing underlying functions and patterns and facilitating decision support are all qualities of a good visualisation and projection method, with a large amount of research having been done on this topic

and several methods being proposed. SOM's are widely used to visualise dimensionality reduction, with its topology preserving property being used to relative mutual relations between data. [8]

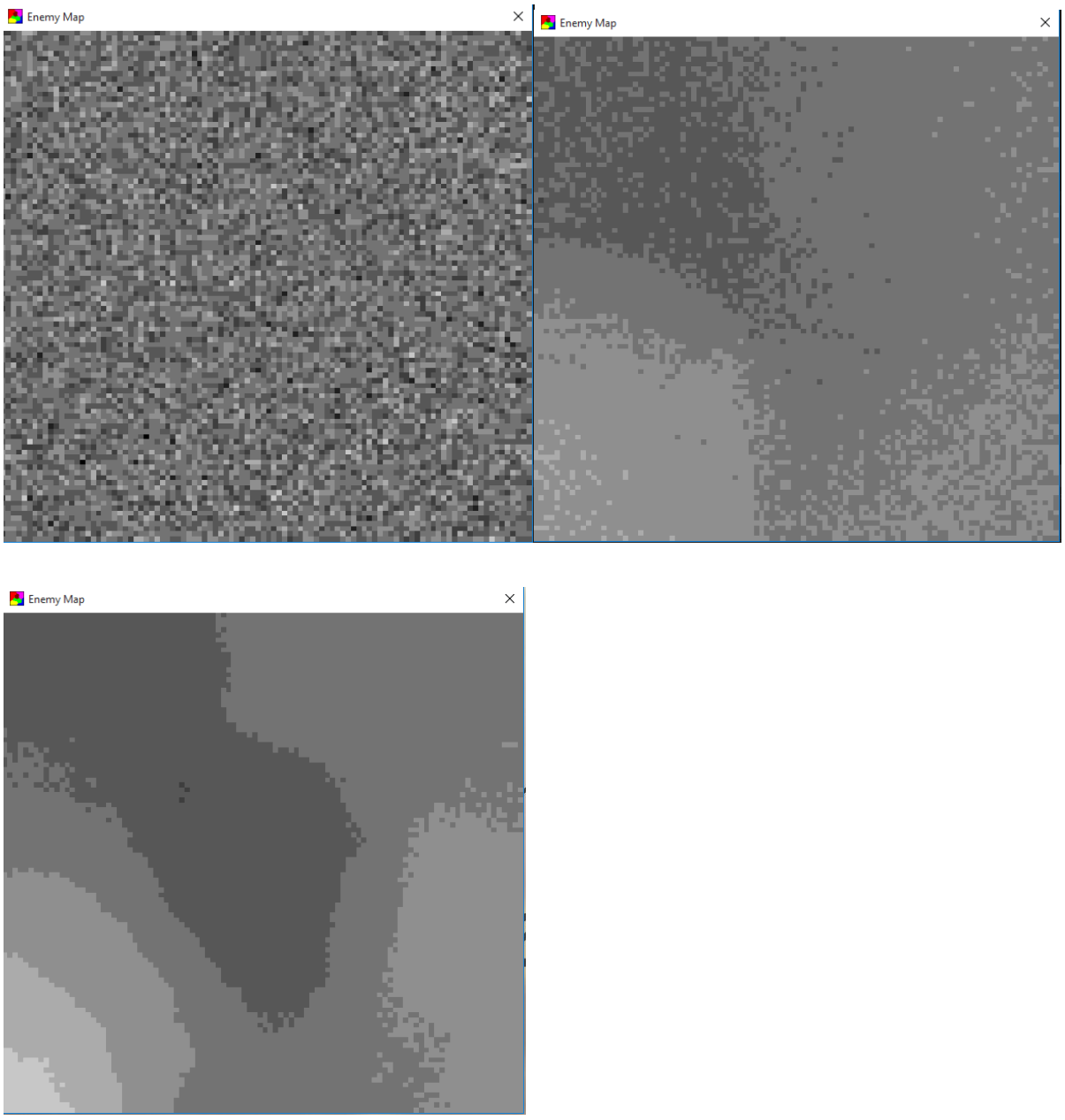


Figure 2.2.4: Visualisation of the organising of data.

# Chapter 3: System Design

## 3.1 Introduction

This chapter covers the design of the system, its implementation and requirements as well as the different technologies used in the project, with reasons for using them.

## 3.2 Requirements

The requirements of the system have been broken up into the requirements of each component, each of which are listed below.

### *3.2.1 Backend Requirements*

The following are the requirements for the SOM, SOM controller class, the NoSQL database and the node class.

- The SOM is required to create a vector containing a predetermined amount of node objects, no more, no less.
- The SOM must be able to find the BMN for a given input vector.
- The SOM must, for each iteration, find nodes within a BMN's map radius and adjust their weights accordingly.
- The values of the training vectors and the weights of the nodes must adhere to the rules of the RPG.
- The node class must be able to adjust its weights.
- The node class must be able to calculate the Euclidean distance between itself and a given input vector.
- The system must be able to connect with and use the NoSQL database.
- The SOM should be able to be stored in the NoSQL database.
- The system should be able to access a stored SOM and retrieve it.

### *3.2.2 Frontend Requirements*

The following are the requirements for user interaction and other front end parts of the system.

- The player stats must be able to be altered.
- The player stats must adhere to the rules of the RPG.
- The player and enemy stats must be displayed.
- The enemy and player characters must have the same number of stats as the nodes have weights.

The front end of the system is not the most important aspect of the project. The idea being that the backend can work with a fully-fledged RPG with ease. The most important aspect of the frontend is that the object being generated in the map as a nodes weights is equal to the object in the game, basically the final requirement in the frontend in the case of this project. The frontend can be as minimalistic or as complex as desired as long as that final requirement is met.

### 3.3 Architecture

The goal of the project is to create a fully functioning self-organising map that will work in the background to generate thousands of possible enemy stat combinations, the project will also connect with a local NoSQL database to store and retrieve the SOM. The project was built in the latest version of unity game development engine. The reason unity was chosen as it provides good and helpful documentation, it is quite simple to use and it is the engine I am most comfortable in using, due to having used it before. The project scripts were all written in C# as it is the scripting language available in unity that I felt most comfortable in using, as opposed to JavaScript which I have a limited experience with. The NoSQL database system chosen for this project was MongoDB. MongoDB was chosen over an RDBMS as Mongo is schema less being a document database where a collection can hold multiple different documents.

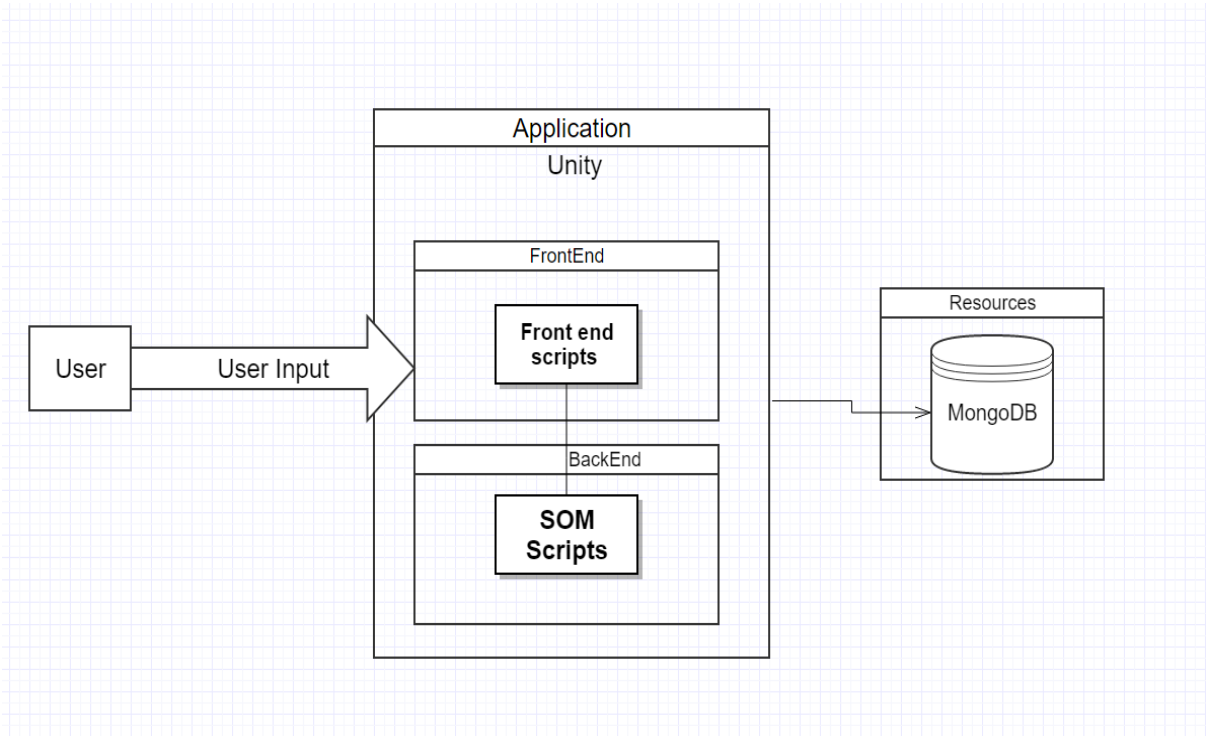


Figure 3.3.1: The high level design of the system

The above image shows the high level design of the system, from how the application built in unity is divided between scripts that work in the back end, the SOM, and the scripts that work in the front end for getting user input and displaying values of nodes etc. The unity project is also connected to a locally stored MongoDB database that holds the SOM once it is created.

## 3.4 Design

### *3.4.1 Low Level Design*

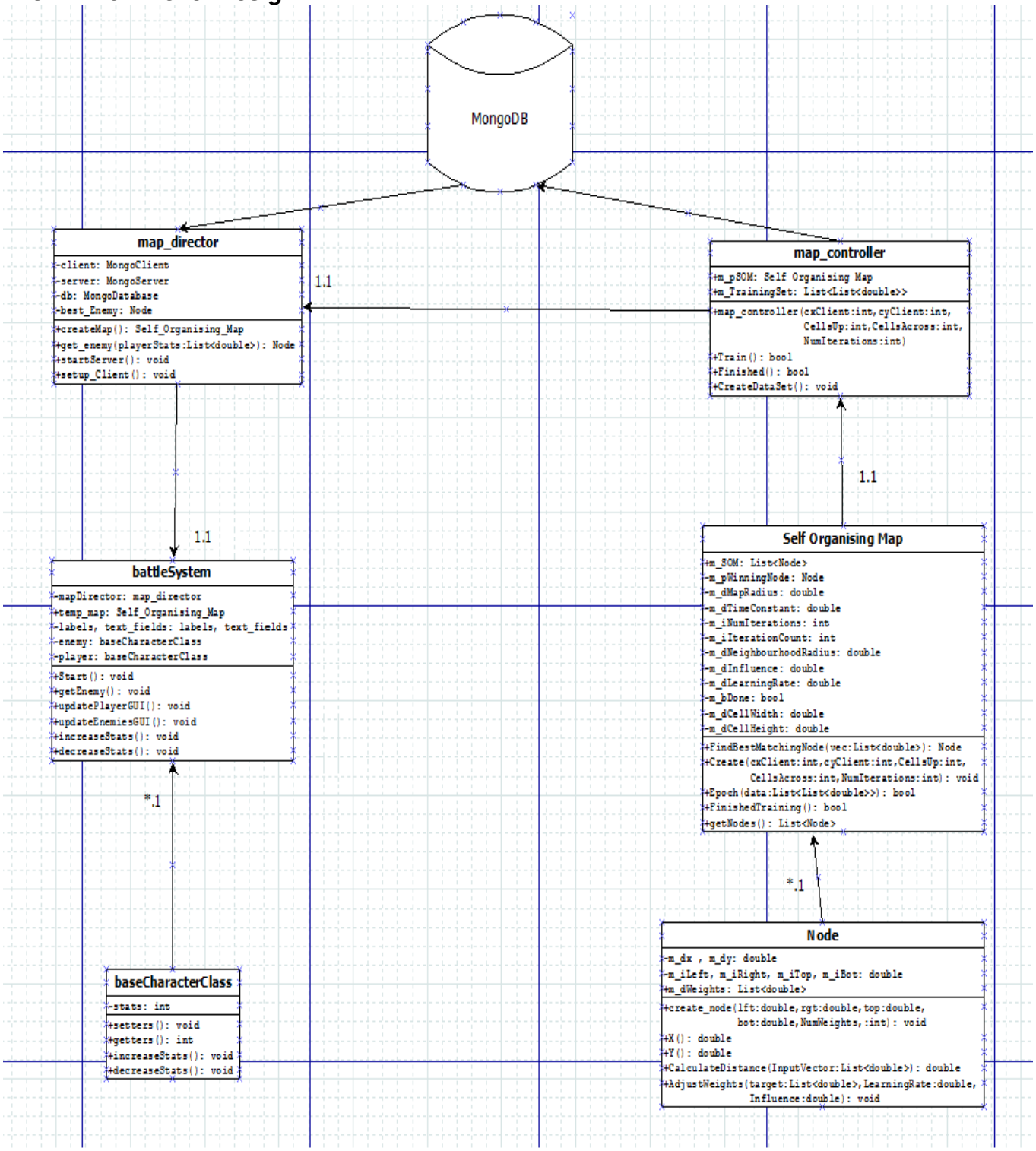


Figure 3.4.1: Low level design of the system, showing the classes and relations

The front end classes of the project are located on the right of the above image while the back end classes are located on the right with the locally stored MongoDB database and server located in the middle. The entire system consists of these 6 classes and the database. The map_controller, Self_Organising_Map and node classes are what come together to form the SOM in its entirety. The map_controller class and map_director class are the classes which access the database. Figure 3.4.1 also shows the relationships of the classes as a whole. The only interaction the front and backend have is through the map_director class which is used to create the SOM on the first iteration and then retrieve it from the database on subsequent retrievals of an enemy. The first class created in the beginning of the system is the battle_system class which then creates 2 baseCharacterClass objects and the map_director, which creates the SOM.

### *3.4.2 Frontend design*

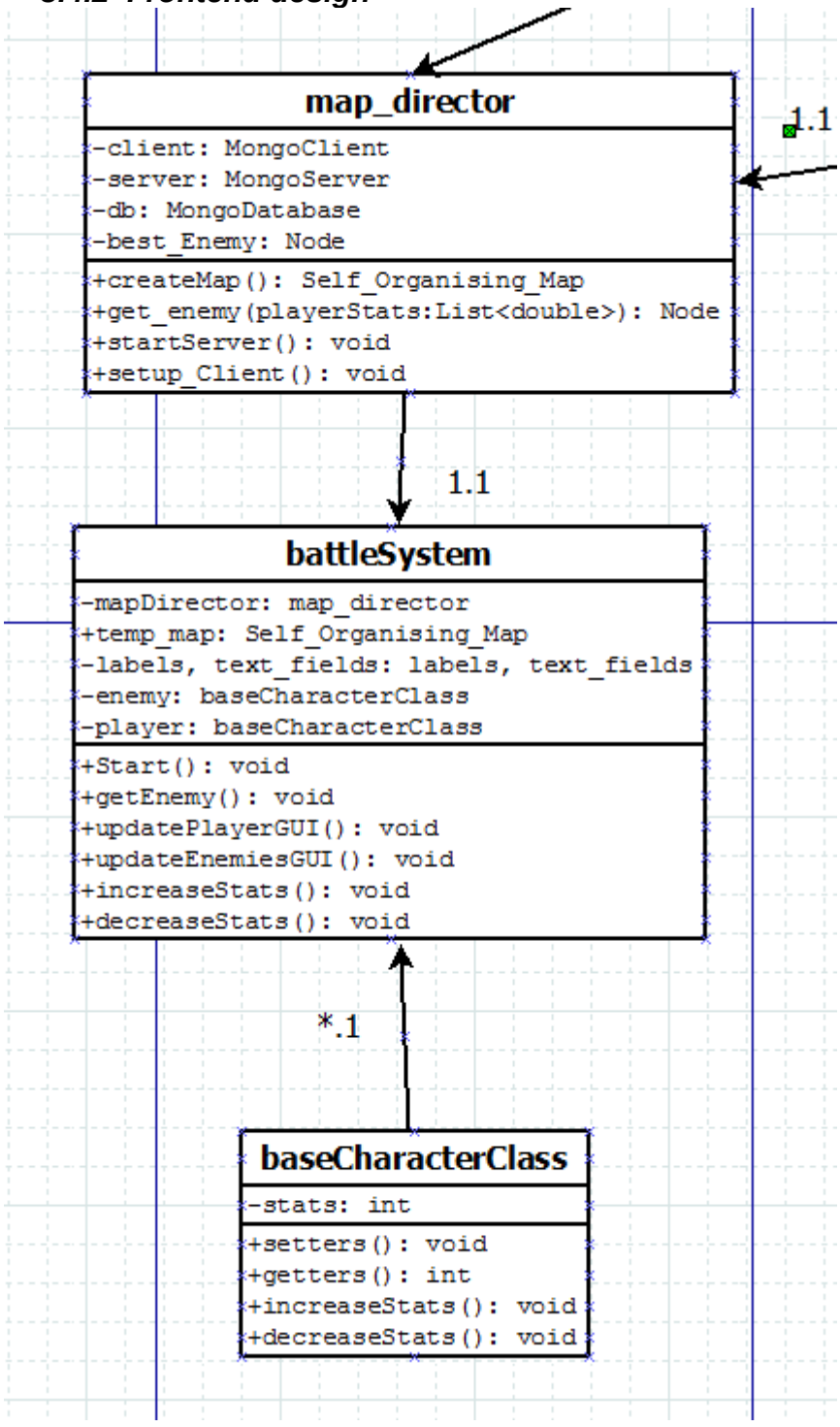


Figure 3.4.2: The classes used in the front end of the system

The front end of the system consists of three classes; the baseCharacterClass, the battleSystem class and the map_director class. The baseCharacterClass is the class used to create enemies and the player, it contains the same number of stat variables as stats used in the RPG. Depending on the role of the character, if it is the player or an enemy, then the stats will be set differently. The players' stats will be set manually

by the user of the program while enemy stats are set based on the weights of the node chosen to be used with this enemy. As for methods, the baseCharacterClass has the usual setters and getters for the stat variables, as well as variable that will increase or decrease the stats, an increase and decrease method for each stat. These increase and decrease methods must adhere to the rules of the RPG when altering the stats, for example if the armour stat is being increased but the strength stat is not high enough for the player to wear higher tier armour then the stat will not be increased until the strength stat is of a sufficient level.

The battleSystem class handles the updating of the GUI's, handling user input for altering player stats and getting a new enemy, instantiating the enemy and player characters' stats. The battleSystem class is the main class of the front end. The battleSystem contains a map_director object which it uses to interact with the back end, creating and retrieving the SOM. The map_director class is the go between class for the front and backend of the system. It contains a map_controller object and uses it to create the SOM. It also uses a MongoDB unity driver to start the mongo server and then use the database for storage and retrieval of the SOM.

### *3.4.3 Backend Design*

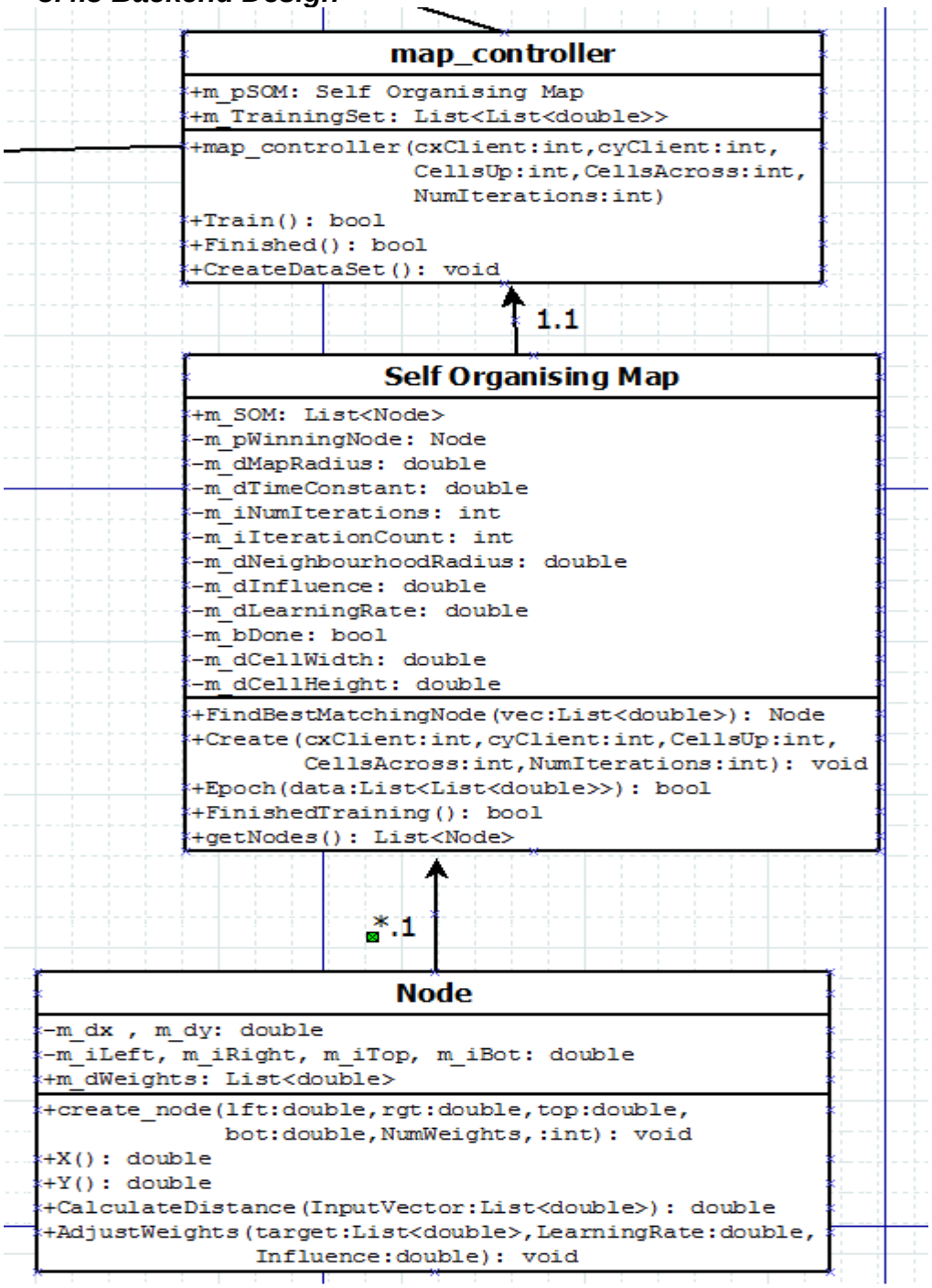


Figure 3.4.3: The classes used in the backend of the system

The backend of the system comprises the whole SOM, which is made up of three classes, the Node class, the Self_Organising_Map class and the map_controller class. The Node class contains the x and y coordinates of the node in the map and a List of its weights. The node is a fairly simple part of the SOM as its only methods are to calculate its Euclidean distance between its weights and that of a vector passed to the node and to adjust its weights, and getters for the x and y coordinates of the node.

The Self_Organisng_Map class is the largest and most complex class in the back end. This class, upon its creation, creates a list of however many nodes that its told to. The class contains the variables that control the number of iterations it does while training, the learning rate of the nodes while training, the initial neighbourhood radius of a BMN and several more very important variables. The Self_Organising_Map class has two very important methods within it, the first being the method to find the BMN of the Nodes, the second being the Epoch method which handles all the iterations and training of the system.

The final class in the backend is the map_controller class. This class contains all the training data for the SOM to use. This class creates and stores an instance of the Self_Organising_Map class and has a method to begin the Self_Organising_Maps training. This class also stores the map in the NoSQL database.

### 3.5 Implementation

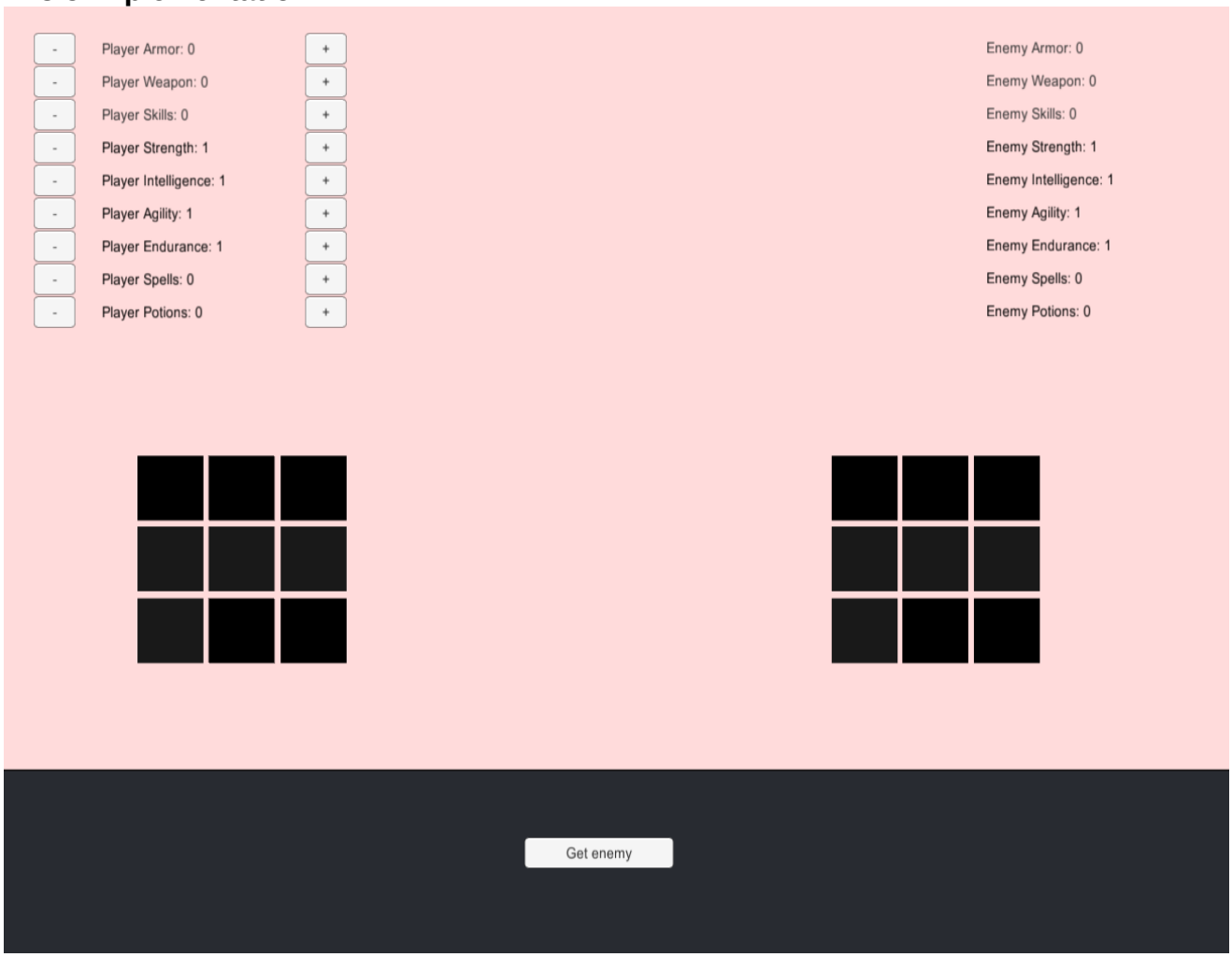


Figure 3.5.1: The main scene of the project.

The system was implemented, using c#, as scripts within unity. The starting point of the system is the battleSystem script which is the first script use upon the starting of the project. When the battle script is started it initialises the player and enemy characters to have the lowest possible values in the RPG. It then updates the GUI elements in the scene to represent this. The methods associated to the increment and decrement stat buttons get the player class to use its local methods that alter its stats based on the rules of the RPG, which checks for the stats value, if it can be increased or decreased based on set conditions such as having an armour stat of four or five requires a strength stat of seven or more etcetera. These stat altering methods in the battleSystem also update the GUIs. The method associated with the get enemy button first gets the battleSystems instance of the map_director to create a map if one does not already exist in the database. Once the map is created it gets returned to a temporary Self_Organising_Map object in the battleSystem. A List of the players' stats is then created and the map and List of players' stats are sent as parameters in the map directors method to find and enemy, which returns a node containing the desired weights compared to the players' stats. The method then sets the enemy stats as the weights from the chosen node and updates the GUI.

The next script in the system is the map_director. The map_directors create map method starts the mongo server using the startServer method and then calls the setupClient method to establish the mongo client with the correct database and collection. It then checks if there is a map stored within the database, if so retrieve it and return the map otherwise it creates an instance of the map_controller and gives it the parameters to create the map such as how many iterations the SOM will run, how many nodes will be in the SOM and so on. The method then calls the map_controllers finished method which returns a bool. If the bool is false then it calls the controllers train method to train the SOM, which also returns a bool when completed. The map is stored in the database and also returned. The getEnemy method takes in a List of the players attributes and a Self_Organising_Map. The method checks the database for a map and if none is present then it uses the map passed to it as a parameter. The method calls the SOMs' FindBestMatchingNode method, passing it the players stats, to find the BMN node to the players' stats which gets returned to the battleSystem. The startServer method just begins a window process with a path to the mongo server

executable file while the setup_Client method sets the client to the local mongoDB and the mongo database to the games database.

The map_controller script is the starting point of the SOM. Upon its creation it creates a Self_Organising_Map object, passing it the parameters passed to itself upon its creation. It then calls the createDataSet method. The createDataSet method creates sever hardcoded training Lists and several hundred randomly generated training Lists and stores them all in a List of Lists. The controllers Finished method just calls the SOMs' FinishedTraining method which returns a bool as to whether the SOM is trained or not. The Train method checks if the SOM has already been trained and if not it calls its Epoch method passing to it the List of training Lists.

The Self_Organising_Map class is the main class in the system and is the most complex. The class' Create method is passed the parameters from the controller class, which are the x and y sizes of the client window, which are used in the calculation of the size of the Nodes and furthermore to find their location in the map and if they are in a BMNs' radius. It is also passed the number of cells in the SOM and the number of iterations to be done. The method firstly creates the width and height of the Nodes using the aforementioned parameters for client size and number of nodes. It then enters a for loop to create every Single Node in the Map, passing the nodes their width and height and the number of weights they will have, which in this case is nine. The SOM adds each node to a List of nodes. The method finally calculates the size of the topological radius of the map and the time constant. As previously mentioned the SOMs' FinishedTraining method returns a bool that is set to true when it is trained. The final method in the SOM is the Epoch method. This method takes in the List of training Lists from the controller. It enters the iteration loop which loops for however many iterations were set. The training Lists are presented to the network at random with the currently chosen training List being presented to each and every node in the network. The chosen training List is passed to the FindBestMatchingNode method that iterates through each node in the network calling their calculateDistance method and passing it the training List. This Node method returns the distance of the node and the training List. This distance is compared to the previous lowest distance and if it is lower than the lowest distance variable is set as its distance while the winning node variable is set to this node. This continues until all nodes are done and the winning node is

returned to the epoch method. The size of the neighbourhood radius is set and a for loop is entered which checks the Euclidean distance of each node in the network and the winning BMN. The distance is checked to see if it lies within the BMN's neighbourhood radius. If so, then the influence of the BMN is calculated based on how close the node is to the BMN and its weights are then adjusted. The method then continues this for all nodes and once all nodes are completed it goes to the next iteration selecting a new training List and repeating until all iterations are done, setting the bool related to whether it is trained to true once it is.

The Node class is the final class in the system. The nodes create method establishes the position of the node in the network and randomly sets the values of the weights in the node. It has two methods, X and Y, which just return the location of the nodes x position and y position respectfully. The node class contains the method calculateDistance which is passed an input vector. It then enters a for loop calculating the distance of each value in the input vector and each of its weights, returning the distance when complete. The adjustWeights method in the node class takes in a list as its target weight values, the learning rate and the influence of the BMN on this node. The method iterates through each weight value and adjusts it using the learning rate and influence, moving it closer to the associate value in the target List.

# Chapter 4: Testing and Evaluation

## 4.1 Introduction

This chapter covers the testing which occurred throughout development of the project, both black box and white box testing.

## 4.2 Testing

### *4.2.1 Black Box Testing*

Due to the rather limited user interactivity with this project this resulted in there being a very low amount of black box testing which needed to be done. The interactive element of the project is the ability to alter the players' stats and to get an enemy with similar stats from the SOM. The first test was to ensure that a new enemy was retrieved when the button to get an enemy was clicked. If the storing of the SOM in the database was not working correctly, then each click of the button without changing the players' stats would result in the retrieval of an enemy with varying stats. If the database is working, then the enemy would be the same as long as the player stats were un altered. The next test was to ensure that the players stats could be altered and that the alteration of the stats did not break the rules of the RPG. The final test was to check that altering the players' stats and retrieving a new enemy would indeed retrieve a new enemy.

#### *4.2.1.1 User Stories*

- The user can adjust the player stats.
- The user cannot adjust the player stats past the lower and upper limits.
- The user can retrieve an enemy.
- The GUI displays update upon a player stat adjustment and retrieval of an enemy
- The enemy chosen for a particular player stats combination will be the same enemy chosen each time those player stats are set

### *4.2.2 White Box Testing*

The development of this project was test driven in order to ensure that everything, especially the SOM worked in the manner it was intended to. Examples of testing include but are not limited to, testing the size of the input vector being used to train the nodes. The size of the input must be the same as the nodes weight vector otherwise the system will break. This check occurs at the beginning of the Epoch method.

Immediately after this check there is another one to check if the testing has already been done before so as not to train an already trained network. The retrieval of an enemy requires a test to check if a SOM is already stored in the database, were this check not there, the system would generate a new SOM each time an enemy is needed. The baseCharacterClass requires several checks in the methods associated with the altering of stats. These tests ensure that the stats do not exceed the maximum limit or go lower than the minimum limit. They also ensure that in order for a character to have high armour they require high strength while the use of a high tier weapon requires either a high strength, intelligence or agility stat. Since the SOM is made up of three interacting classes, the controller class, the Self_Organising_Map class and the node class, there needed to be a number of checks to ensure that each part of the SOM was working correctly with no issues. An example of this is the List of nodes in the Self_Organising_Map class. If after the networks creation, the number of nodes in the List does not match the number of nodes that the system is set to create then something is wrong with the generation of the nodes.

### 4.3 Evaluation

Due to the limited nature of user input with this particular project and the low number of interacting systems, mainly just the front end and the self-contained SOM system, there is not much room for error apart from some areas where crucial checks need to be made. The system is built effectively with no wasted methods or code, that every part of a method can be reached. The system is easily maintainable and modular so the SOM can be altered to add more methods which may interact with it. The SOM in particular works well with zero issues in terms of bugs. The SOM is the main area of the project which required testing to ensure it worked perfectly as it is the core of the project. If the SOM contained any bug which could alter the training, organising or any part of the SOM then the project would have failed.

# Chapter 5: Conclusions

## 5.1 Introduction

This section covers the overall conclusions of the finished system, what changes could be made to the system and any issues that are or were raised within the system as a whole.

## 5.2 Reflection

After working on this project for several months and doing much research on SOM's I believe that they have a wide range of benefits and uses. In terms of procedural generation, specifically of enemy characters, they are extremely useful especially since they organise and cluster the enemies. This clustering allows for the retrieval of several enemies from the map with similar ranges of stats if needed. However apart from numerical values, SOM's aren't tolerable of different data types and need to be used with these numerical values which may be an issue in terms of procedural generation. For example, if a user wishes to procedural generate thousands of instances of a class object which has a non-numerical variable in the class then they would have to work out a work around as to have the non-numerical value be set using a numerical one or use a different way of procedurally generating them. The SOM also has a major issue with stability of its models as seen in Appendix A. This is due to the fact that the weights of each node are initially set to a random value, meaning the majority of the nodes may begin with a high value and even after the SOM was trained and the nodes weights adjusted there may be very few nodes with low values, and vice versa where the nodes may be initialised with low values. Generating an SOM is also quite computationally expensive, however this would only be an issue if the SOM is being generated at runtime and is not pre generated and stored in a database. Given more time with the project id have liked to add more of an interactive element to it, and have an actual battle system to test how these procedurally generated enemies faired against a human controlled opponent and to see if they really posed a threat.

## 5.3 Recommendations

I would recommend SOMs' be used in the generation of RPG enemies, or any non-complex object that would require generation since SOMs' preserve the topology of data and cluster it as well even though they do lack stability. However, for more complex types of procedural generation I would recommend using a different method or even a different neural network, unless a work around is used with an SOM. As for

SOM’s in general they are an extremely useful tool for procedural generation since the use a form of unsupervised learning. This unsupervised learning is great for procedural generation where there is no correct answer just different types of answer. As for the engine used in this project, unity is a very good game engine to use and is very intuitive and user friendly to work with. Even in terms of connecting with the Mongo database it was quite simple and intuitive to implement.

### 5.4 Final thoughts

SOM’s are incredibly powerful tools to be used for procedural generation and A.I. learning. Since they use unsupervised learning they do benefit procedural generation which typically does not know what the correct thing to be generated is. However, SOMs’ still require more research in order to improve on or remove this instability in the models which is causing, in terms of procedural generation, a lot of issues as can be seen in Appendix A. Once this issue is resolved then I feel that SOMs’ should be used more often in the gaming field.

**References**

[1] D. M. Bourg, G. Seemann, "Neural Networks" in AI for Game Developers, 1st ed., Sebastapol, O' Reilly Media, 2014.

[2]J. Noyes, *Artificial intelligence with Common Lisp*. Lexington, Mass.: D.C. Heath, 1992.

[3]S. Guthikonda, *Kohonen Self-Organizing Maps*, 1st ed. Wittenberg: Wittenberg University, 2005.

[4]C. Stergiou and D. Siganos, "NEURAL NETWORKS", 2008.

[5]T. Kohonen and T. Honkela, "Kohonen network", *Scholarpedia, vol. 2, no. 1, p. 1568, 2007*

[6]T. Germano, "Self-Organizing Maps", *Davis.wpi.edu*, 1999. [Online]. Available: http://davis.wpi.edu/~matt/courses/soms/. [Accessed: 09- Mar- 2016].

[7]J. Bullinaria, "Self Organizing Maps: Fundamentals", University of Birmingham, 2004.

[8]H. Yin, "The Self-Organizing Maps: Background, Theories, Extensions and Applications", in *COMPUTATIONAL INTELLIGENCE: A COMPENDIUM*, 1st ed., Berlin: Springer-Verlag, 2008, pp. 715-754.

[9]R. Baltimore, "An Analytic investigation into self organizing maps and their network topologies", Undergraduate, Rochester Institute of Technology, 2010.

[10]G. Orr, "Neural Networks", Willamette University, 1999.

[11]A. Paiva, "Competitive learning, clustering, and self-organizing maps", University of Utah, 2008.

**Glossary**

*Topology*: The mathematical study of properties that are preserved through deformations.

*Vector/List*: A method of storing data. Can only store one type of data and have no set number which they can store.

*Procedural generation*: The creation of data algorithmically as opposed to manually.

*C#:* A programing language and one of two languages used in the unity engine.

*Parameter*: A variable that must be given a specific value during the execution of a program or a procedure within a program.

*Script*: A program or sequence of instructions that is interpreted or carried out by another program rather than the computer processor.

*Database*: An organised collection of data.

*Algorithm*: A self-contained step-by-step set of instructions

*Maxima/Minima*: The largest and smallest value of a function either within a given range or the entire domain of a function.

**List of Abbreviations**

ANN: *Artificial Neural Network*

SOM: *Self Organising Map*

BMN: *Best Matching Node*

VQ: *Vector Quantisation*

GUI: *Graphical User Interface*

*RPG: Role Playing Game*

*AI: Artificial Intelligence*

# Appendix A: Raw data.

## A.1 Lowest node weights.

The following table shows the values of twenty different nodes taken from twenty different generated SOMs. Each node is the node with the smallest possible weights within the network.

| Armor | Weapon | Skills | Strength | Intelligence | Endurance | Agility | Spells | Potions |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

Table 5.4.1: The smallest possible node weight value

| Armor | Weapon | Skills | Strength | Intelligence | Endurance | Agility | Spells | Potions |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 2 | 2 | 3 | 4 | 0 | 3 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 3 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 3 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 2 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 2 | 1 | 2 | 0 | 0 |
| 0 | 0 | 1 | 4 | 1 | 1 | 2 | 0 | 0 |
| 0 | 2 | 0 | 1 | 1 | 1 | 2 | 1 | 0 |
| 0 | 0 | 0 | 3 | 2 | 2 | 4 | 0 | 0 |
| 0 | 1 | 0 | 2 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 3 | 4 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 2 | 1 | 0 |
| 0 | 0 | 0 | 5 | 1 | 2 | 2 | 0 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 3 | 1 | 3 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 3 | 0 | 3 |
| 0 | 0 | 0 | 2 | 3 | 2 | 3 | 0 | 0 |
| 0 | 0 | 0 | 3 | 2 | 2 | 1 | 0 | 3 |
| 0 | 1 | 0 | 2 | 2 | 2 | 1 | 0 | 0 |
| 0 | 1 | 0 | 2 | 1 | 2 | 2 | 0 | 0 |
| 0 | 0 | 0 | 4 | 1 | 2 | 1 | 0 | 3 |

Table 5.4.2: Raw data of the lowest valued node in 20 different generated SOMs