Design, Implementation and Research into

Evolutionary Algorithms used in 2D Games Design

James Copping

BSc in Computer Science

03/04/2019

SCHOOL OF COMPUTING AND MATHEMATICS

Keele University

Keele Staffordshire ST5 5BG

1 Abstract

In this report feasibility of designing and implementing a unique game framework is explored. The implementation will use Evolutionary algorithms in order to produce level-based content which increases in difficulty. The project is split into three sections, each clearly define the goals of the project. The aim is to provide an understanding in the workings of procedural content generation while using neural networks and co evolution ins a novel fashion.

It is evident there are many approaches to the issues raised in the report and there are many different uses for the design and techniques that are explored in the paper. It is reasonable to say that for a small game it can be appropriate to implement similar systems to the ones in use here. This can also be effective at a larger scale and the approach is scalable to many problems in this area.

2 Table of Contents

3 Table of Figures

4 Introduction

The problem that is being addressed in this report is the approach and feasibility to generate levels automatically in a 2D platforming game trough the use of genetic algorithms and co evolution. Can levels in a game be generated using these techniques? Is it appropriate and reasonable within the context of the game to do so? Then what else can be done with this approach? How can this be utilised to gain an edge over traditional level design?

There have been many different pieces of work done in this area in detail. The concept of procedural content generation (PCG) has sparked an interest in Computational Intelligence (CI) communities and research over the last few years. “Procedural content generation (PCG) refers to the creation of content automatically through algorithmic means” (Yannakakis, 2011). In the paper titled The Mario AI Competition: Level Generation Track it is stated that “A key concern for many commercial game developers is the spiralling cost of creating high-quality content (levels, maps, tracks, missions, characters, weapons, vehicles, artwork etc) for games.” (Shaker, 2010) This means that there is a need for alternative methods to create these assets, the increasing standard of games in regards to graphics, design and content is causing this shift in the industry to look towards more effective ways to generating content such as levels. Therefore, it is an important area of research in modern computing and attractive to both big and small game development companies.

The aim of this project is to demonstrate the design and implementation of a prototype Game *framework* which will utilise this idea of PCG and user driven design to produce 2D platforming levels with a genetic algorithm approach. These levels also need to be engaging, fun or challenging in some way. This being the case is it possible to push a co evolutionary algorithm to generate increasingly difficult levels and if so how effective is this? “Genetic algorithms instead allow developers to specify desirable level properties in a top-down manner, without relying on the specifics of the underlying implementation. However, any effective fitness function for automated level creation must correctly identify the levels that are “fun.” To this end, a model of what precisely constitutes a fun level must be developed.” (Sorenson & Pasquier, 2010) This idea of measuring fun from a generated level is hard to define and for the scope of this project it is reasonable to simplify what this means in the context of this game. The increasing complexity and difficulty of the level will be defined as fun.

What will be the product of report? The report will follow the design and implementation of a game framework from which a neural network will be able to simulate a player learning a level. From this the co evolution of the level content can be built.

The project is therefore split into three distinct stages of development:

1. Game Framework
2. Neural Network Integration
3. Co-Evolution of Levels

The result of the three stages will be a prototype for a game that may demonstrate how the techniques can be applied to a real game.

5 Report Body

Development preface

The problem that was being addressed requires a hybrid approach to the development of the solution. The *solution* or more appropriately, the proof of concept had to be divided into three separates stages. Each stage incorporating their own cyclic design, implementation and testing flows. After each stage a project review would take place to ensure the integrity of the previous stages that the top-level implementation was built upon. Essentially providing a feedback loop to each stage as the project changed and the requirements altered to fit the new design approaches.

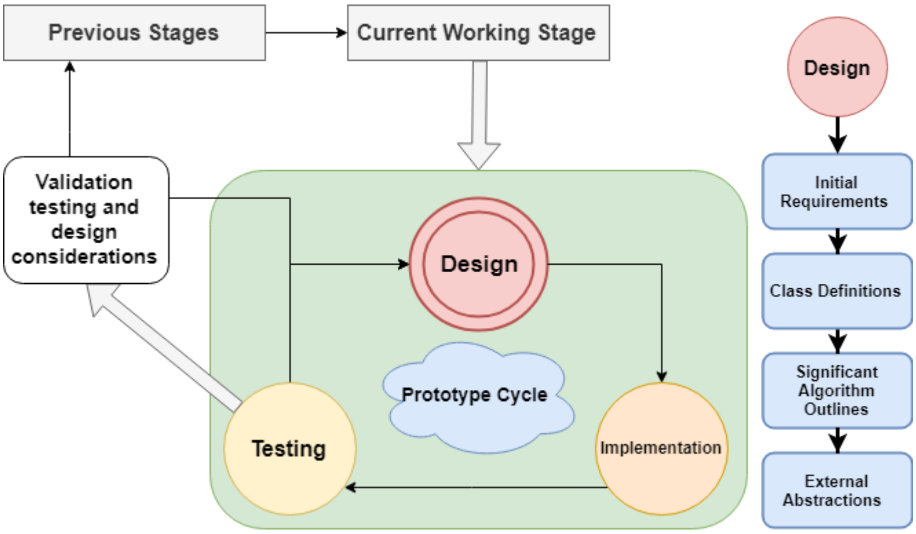


Figure 1 Hybrid Development Cycle. Prototyping and fallback design and shown on the right internal design flow.

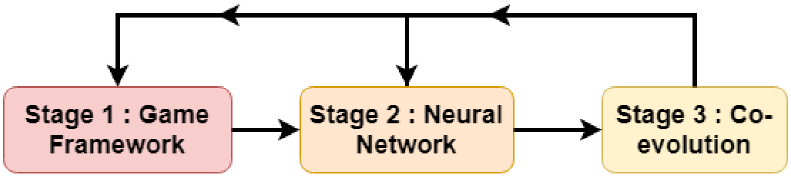


Figure 2 Three Stages of Development

The internal developmental cycle consisted of the topics shown in figure 3 above.

The process shown in figure 1 was chosen based on the initial structure of the project. This seemed most appropriate to tackle the issue. As the questions raised previously show that there are two distinct areas to explore and research. This would therefore require a bespoke foundation to create testing environments and perform analysis on the data that were collected.

The following sections of the report will walk through each stage, including their independent development cycles and how each would feedback to the earlier stages to provide the correct alterations needed for the subsequent stages. The initial stage was the foundation of the project. This was the Game Framework for which each upper stage would function and provide support for creation of each platform including testing, training, validation and generation of populations etc.

This section will not include everything to do with the steps of the cycle that were carried out during stage 1. This is because the development process of the framework is not the focus of the project, despite it being so board in the programming and functional sense. However, the key and interesting points will be discussed. This is still an important and large part of the project that took most of the time. Which is why is will be cut down and simplified.

Stage 1 Game Framework

The first stage of design requires a platform for the testing and some prototype gameplay to occur. The basic structure for the framework will be shown and described in the list below. This will demonstrate the core modules that will be developed in this section. This was not implemented in any strict order, although some modules/classes are dependent on others.

Initial Requirements

- Main Game loop
- State machine and states
- Input handling
- Asset management (sprites, fonts, texture sheets, animation)
- Simple GUI system (Buttons and Labels)
- Level System
- Entity System
- Player Controller
- Animation Controller

This list outlines the base function that are to be implemented into stage 1. Each module is fleshed out in a different fashion and will be revisited in later sections as project alters around the framework. The two largest parts will be the player controller and the level system. However, the critical systems in this stage are the main Game Loop and the State Machine/States and extra care was taken to ensure the integrity of these systems.

For the programming task c++ will be used in close conjunction with *SFML* 2.5.1. This library is a Simple and Fast Multimedia Library, this will assist in areas such as Images/textures, user input, clocks, simple shapes and vectors. This library is being used in this project under the zlib/png licence. The API Documentation can be found here (Gomila, 2018). Most notable the SFML/Graphics.hpp was used extensively.

A side note, all the art and animation used in the game was created by the author of this report and some of the texture sheets will be found in the appendix.

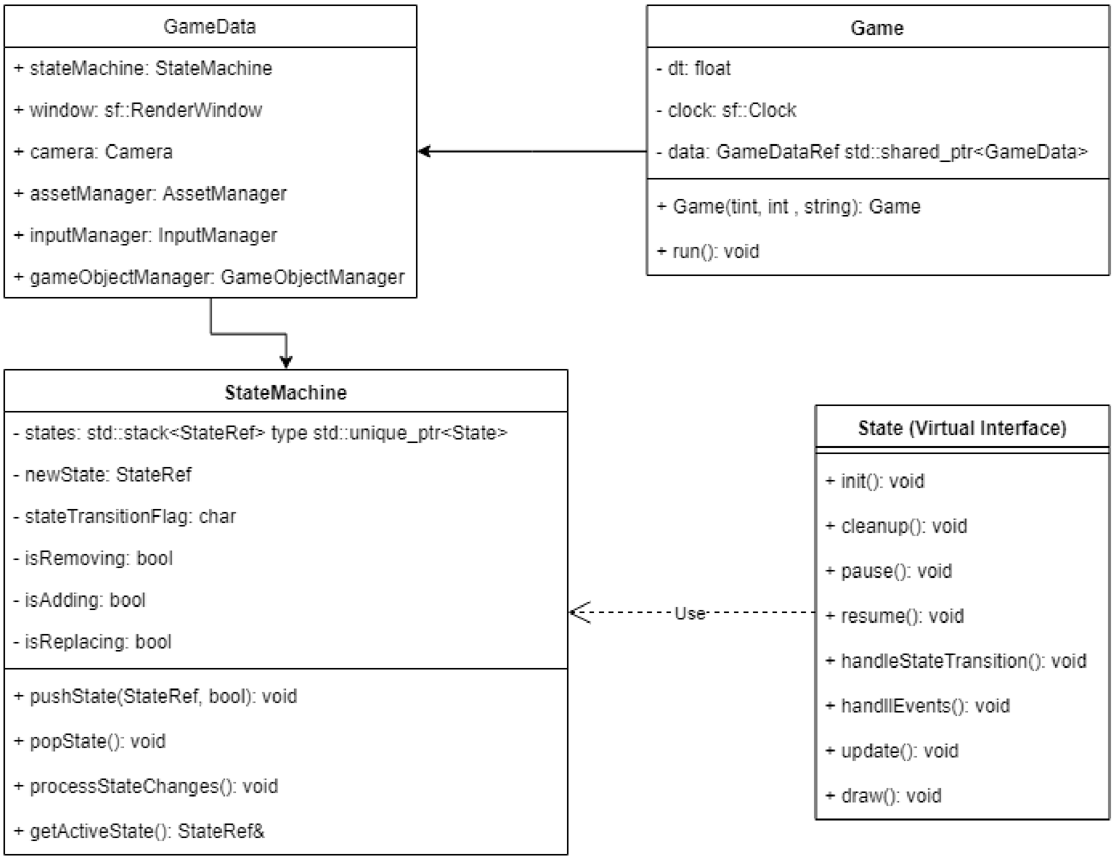


Figure 3 Basic Class Diagram of the Game, StateMachine and State Classes

Figure 3 shows the relationship of how the Game class has a private member vibrable called data. This is a shared pointer which is of a struct called GameData which is where all the information about the game is found and pulled from during run time. GameData includes instantiations of classes such as Camera, AssetManager, InputManager, GameObjectManager and as shown here StateManager, which intern depends on the State class.

The game class has a private method called run which is called at runtime after the class has been constructed. The run method [10.1.6] controls the flow of the states and processes the update and draw calls of the state which is currently on the top of the stack in the game's data member. This is how the game can transfer from one state to another, if in the current state another state is pushed onto the state machine's stack at any time, the game will process the changes therefore rendering and updating that state instead, the pure virtual functions in the state interface are the entry points to each state.

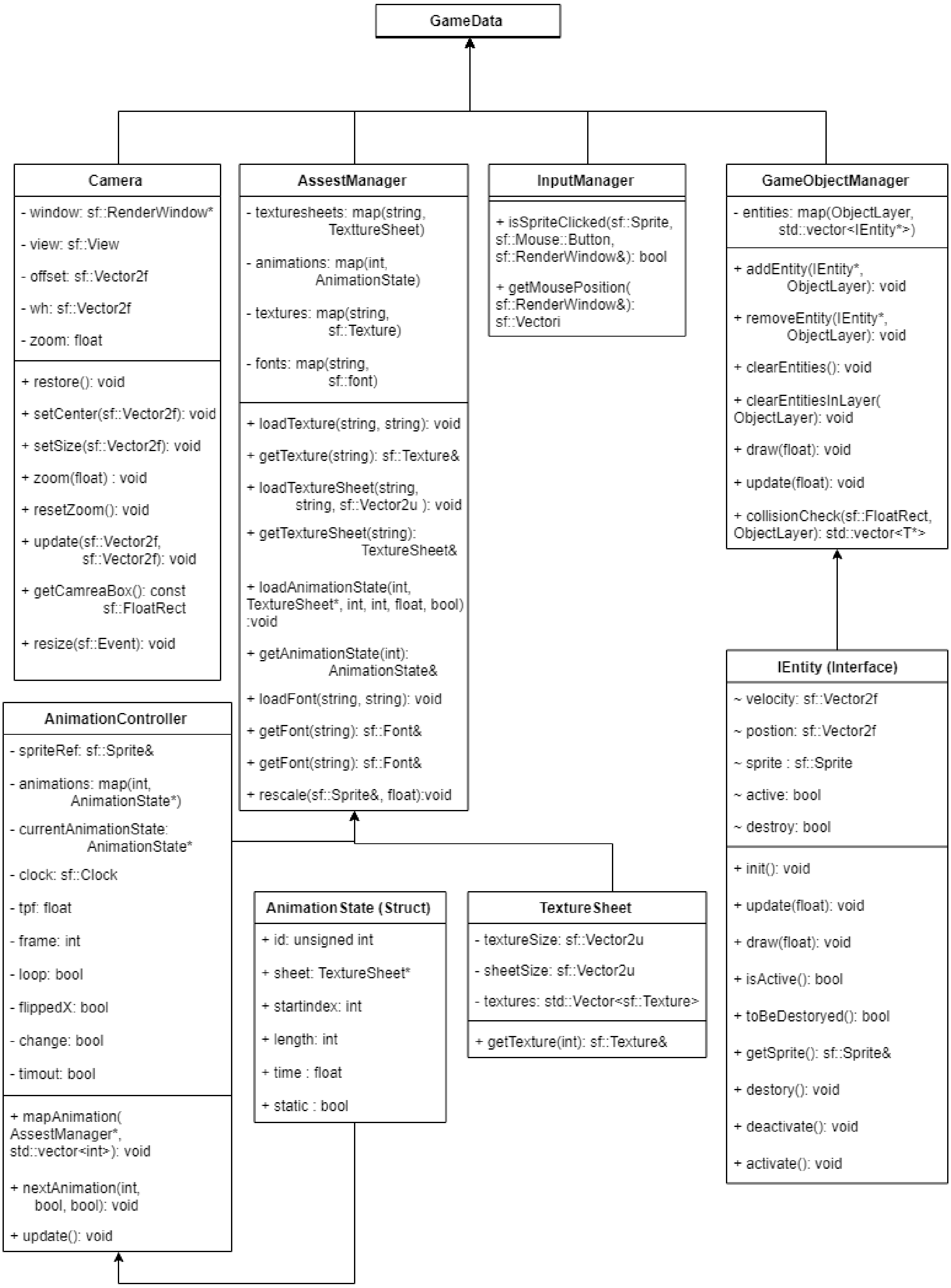


Figure 4 Rest of the GameData Class Dependencies, as can be seen a semi hierarchical structure is designed to to invoke a layer of abstraction and each module in use.

This object-oriented design approach to the framework allows for growth. This means that as the programmer multiple different states can be created to fit any need. The basic layout and flow of the states for the game is as follows: splash screen, main menu -> [gamestate, training, validation, exit etc.]. The main menu screen is populated with buttons that when activated create the new instance of the desired state then pushes it on to the stack to be processed.

All the states are extensions of the pure virtual interface class *State* shown in figure 3. The splashscreen state is utilised to load in the various textures used throughout the game into the assetManager. The assetManager acts as the public pool of all the textures that are available. When creating an entity or a tile (this will be covered in the Level class section) the appropriate texture is referenced to from which the sprite is generated and attached to the object. The gameObjectManager acts as a pool of entities. This pool can be manipulated by the current state, either to add or remove objects from the pool from a given layer. This class also handles interlayer collision. The code for that function can be found [10.1.7]. This class splits up the entities into the layer they are assigned at construction. This allows the layers to be drawn in a specific order, to ensure that the correct display of the level and the entities is drawn. This also opens the way for future implementations of gameplay elements.

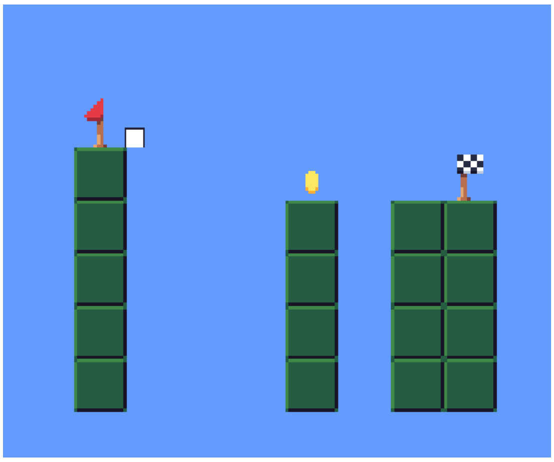


Figure 5 Simple Example Level Generated, with the player entity (white box) and a coin entity (gold blob)

This brings us onto the Level Class and the Player Class. These two classes are the majority of stage 1's gameplay functionality.

The initial implementation of this class had the base requirements of:

- Made up of an array of *tiles* that can be loaded from an external file
- Levels require start and finish
  - Checkpoint system
- Levels can be made by stitching to levels together
- Sections of levels need to be mutable
- Collision detection for the tiles
- Optimisation of collision with tiles and rendering
- Levels can be randomly generated
- I/O of the tile maps.
- Game state

There was may design considerations that had to be considered at this point. The Level needs to be made up of an array Tiles that all had their own position and sprite/hit box, to provide the level with solid ground to the player to collide with. The main tilesheet art that is used in the project up to this point can be found at [10.1.13], bear in mind that not all the textures are used and some are just used in testing, or could not be implemented due to time constraints. Design considerations for The Tile class come under how the tile will be identified, so that a tile map of the level can be stored and when read later, the program know what each tile is and where is should be. Some tiles are going to be solid and some are not going to have any collision. Therefore, it was decided the index position of the tile in question would be used to identify the type of tile that is in use. This also means that to write the tile map to a file, all that is needed is the tile id in a one-dimensional array which can be encoded into the file given the levels height and width. This is an example '.tilemap' file.

```
61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61
61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61,61
61,61,61,61,61,61,61,61,00,00,61,61,61,61,61,61,61,61,61,61
61,61,61,61,61,61,61,00,00,00,61,61,61,61,61,61,61,61,61,61
61,61,61,61,61,61,00,00,00,61,61,61,61,61,61,61,61,61,61,61
61,32,61,61,61,00,00,00,61,61,61,61,61,61,61,61,61,61,33,61
61,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,61
61,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,61
60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60,60
```

Figure 6 Validation Level 5 encoded in the tilemap file format

The file shown above includes the tile id's ranging from [00-63]. Each id points to a different texture in the main tilesheet [10.1.13]. The id corresponds to the index of the tile in the sheet, it being 8x8 tiles and 16x16pixels means is room for 64 tiles. Tiles can not therefore be animated at this point, but the ground work to tiles to be interactable is there so in future tiles can change during runtime of the game, which could be used to create a dynamic level that alters based on interaction with the player or reaction to another entity.

All this preamble to make an instance of a level independent is so that given two levels you can take the file definitions of both and *stich* them together to create one larger level. This created an interesting programmatic problem, as given two level tile maps, which have different heights and widths. Just mashing the two tile maps into each other causes issues. For example, the level might finish at a lower y value relative to the start of the next level, if the gap in the finish flag and the first checkpoint of the second level in the y dimension is more that +1 then the player will never be able to complete the level. This then requires the levels to shift based on the positions and heights of the flags. An example of two levels being stitched is shown below. The code snippet for the stitching of levels can be found here [10.1.8]

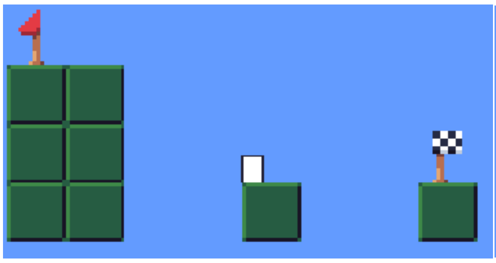
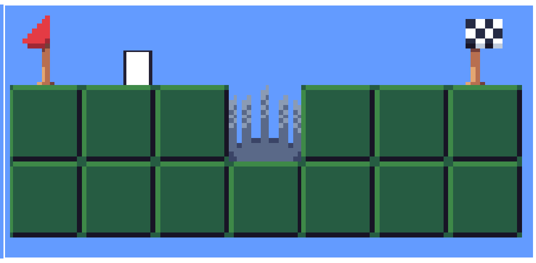



Figure 7 Validation Level-6 on the right

Figure 8 Random Level Generated with Height Map functions on the left

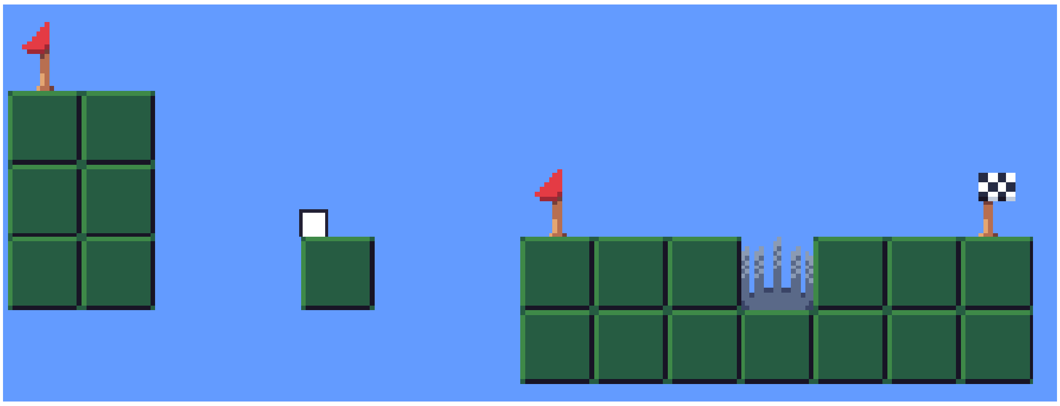


Figure 9 The combination of both levels into one, the level will function as intended and the finish flag of the first level is now a checkpoint.

From this example it might be hard to imagine having a recursive action that adds sections of randomly generated levels to the end of a long level stitching them into one longer level. Later, in the Co-Evolution section, the splitting of larger levels into sub sections that can be altered by the mutation and crossover will be overviewed.

For each level they have a corresponding entity map also. However, when splitting and stitching levels the entity map is ignored for the time being and gets truncated as a result. For the purposes of simplicity regarding the later stages this wasn’t implemented.

The levels that are being shown here are being generated at run time. This is done by a creating a height map of noise from a changing delta value. This is then translated into the columns of the level. After which traps and pit falls are added, while making sure that the level is still completable. The concept of completable will be coved when looking at the player calls as it involves the physics and the controls that the player must work with.

The object instantiation of the Player Class is the body which the user (or later, the neural network) controls to traverse the 2d platforms that are created (Levels) and reach the end (finish flag) without falling off the platforms or touching a trap. The Player must, therefore, be able to collide with the tiles and entities within the current level. The player will also need to be able to go left and right at a given speed and then be able to jump.

The velocity of the player is altered by methods that flip and add magnitude to the direction vector. This is then added to the players velocity each frame and the position is calculated after checking to collision in the x then the y. Updating the positions ‘xy’ independently allows the players hit box to slide on the surface of the tiles if the player has a given velocity. The code snippet for the player/level/tile’s collision detection can be found here [10.1.9]. The player also controls its current state and can alter this with functions such as: nextLevel, restart, respawn.

Once each of these interlinking classes were implemented. The GameState was created to act as a prototype and testing ground for play testing. By play testing and fixing bugs on an elimination basis this would ensure that the functionality that was desired, was achieved.

State 2 Neural Network and Evolutionary Algorithm

For this stage it was decided that primarily the functionally of the neural network framework needed to work before implementing and connecting the dots with the game framework from stage 1. Therefore, a separate project to test the design was the first step. For the purposes of the project the only changes that will be made to the population of networks is the alteration of weights, no node function changes or topology changes. This will be expanded upon in the conclusion.

The approach for this design is to keep all the networks fully connected. By having a topology as an input, several networks will be generated with this layout. The Neural Network Class is abstracting the format of a fully connected network into an array of 2D matrixes. Each index of the array will denote a layer of the network. For example, if the topology of a one hidden layer network is 20 inputs, 10 hidden layer nodes, 4 output nodes. Then the array would have a size of 2 the first matrix at index 0 being a

```
Function matrixOuput takes a matrix, and input array and an enum for the node function
to apply to the sum
Initialise output array
Initialise I,J to zero
While I is less than the number of rows in the matrix
        While J is less than the number of columns in the matrix
                Output at index J add
                          (weight at matrix index [I, J] multiplied by input at index I )
        End
End

Foreach value in output
        Switch on nodefunction
                Case sigmoid
                        Output equals function sigmoid(output, sigmoid param) End
                Case hardlimit
                        Output equals function hardlim(output , hardlim theta) End
        End
End

Return output
```

Figure 10 pseudocode function for the output array of a given matrix with input array

10x20 and the second layer being a 4x10. The layout of the matrix is such that it makes the problem of feedforwarding the values easy.

From the pseudocode (Figure 12), if given an initial input, this function could iteratively be loop through for each layer of the network, using the output of one layer for the input of the next.

In order to train a network to solve a certain problem an evolutionary algorithm is used to generate increasingly better populations of the network. the requirements for this are as follows:

- Perform mutation and crossover functions on the networks to generate new populations
    - Networks must be able to be converted to a chromosome of weights
    - Select networks for mutation and crossover based on the pinwheel selection of the population
- Sort a population of networks, given their individual fitness.
- Evaluate networks fitness ratio, used in the selection of parents
- Write a file which contains all the training data for the generations
- Training and Validation States

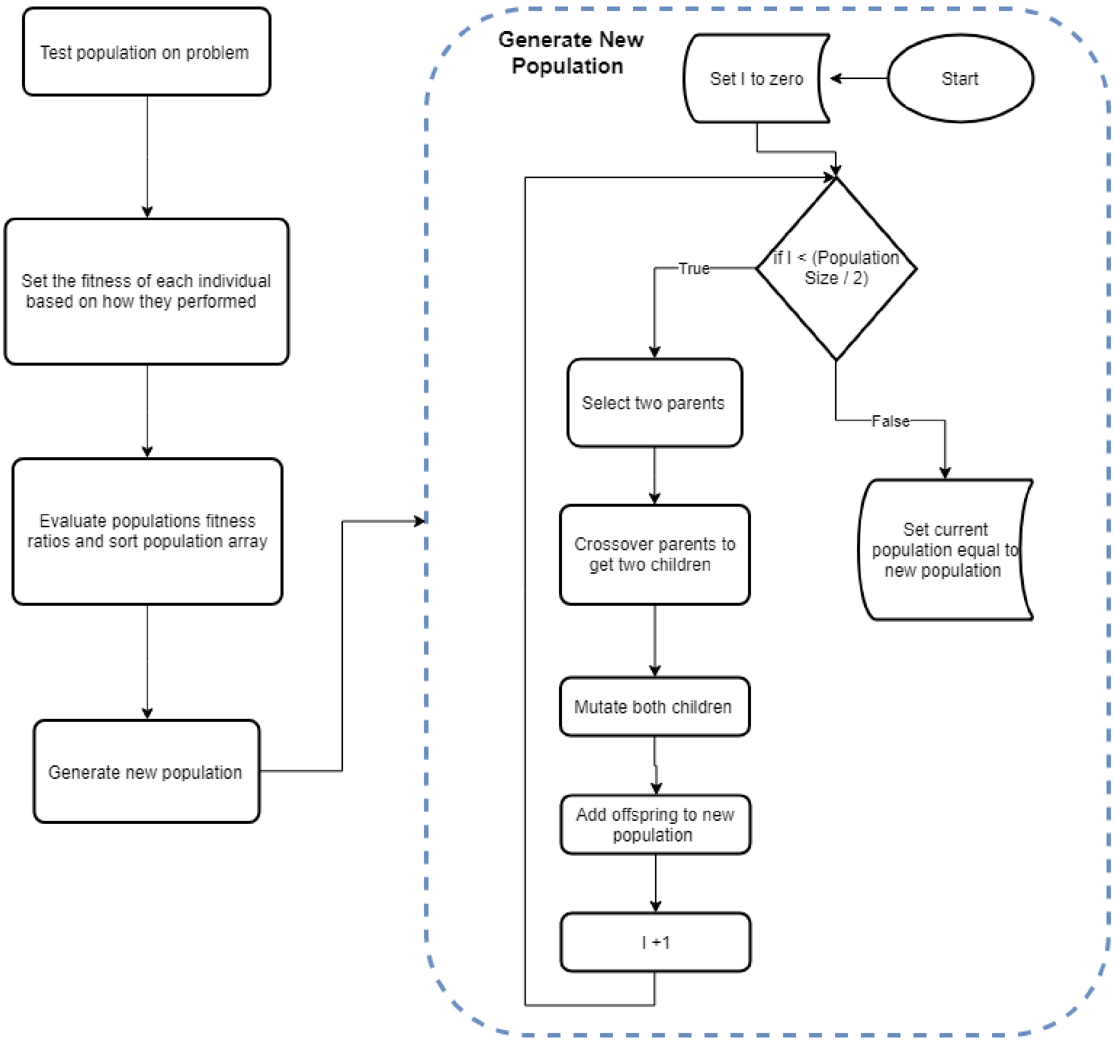


Figure 11 Flowchart for the training cycle of the population

Details about the crossover and mutation functions will be cover in a later section as this is relevant to the training.

The standard genetic algorithm given in (Man, 1996) is the basis of the approach that was taken. Utilising the Roulette Wheel Parent Selection and basic crossover and mutation functions to alter the chromosomes of the two parents to produce variant off spring.

A test problem was given to the evolutionary algorithm side, which was as follows.

For a given network, find the network with the weights that most closely resemble an array of floating-point values. So, if the topology of the test network was [2, 2, 2] then that would have (2x2) + (2x2) connections which is 8. The target array would be 8 iterations from 0.0f to 1.0f. (0.125, 0.250, 0.375 … 1.0f) this would be converted to a chromosome and the weights of the network would be too. Then a comparison would be made for each pair and the fitness of the network was given by how close each value was.

```
0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875,
Generating 120 networks.
 ---please wait---
-!!-DONE-!!-
Starting...
---[15] of 10000 (Fittest Network Percentage: 61.0851%) || (Average Generation Percentage: 56.4167%)
```

```
---[103] of 10000 (Fittest Network Percentage: 80.975%) || (Average Generation Percentage: 69.7424%))
```

```
---[408] of 10000 (Fittest Network Percentage: 96.4923%) || (Average Generation Percentage: 92.6545%)
```

Figure 12 this image shows the progress of the genetic algorithm that is searching for the network with the set of weights closest to the string at the top in the first image

With a target percentage fitness of 95% accuracy produced the following connections of the 2,2,2 network (# symbol indicates a layer of connections)

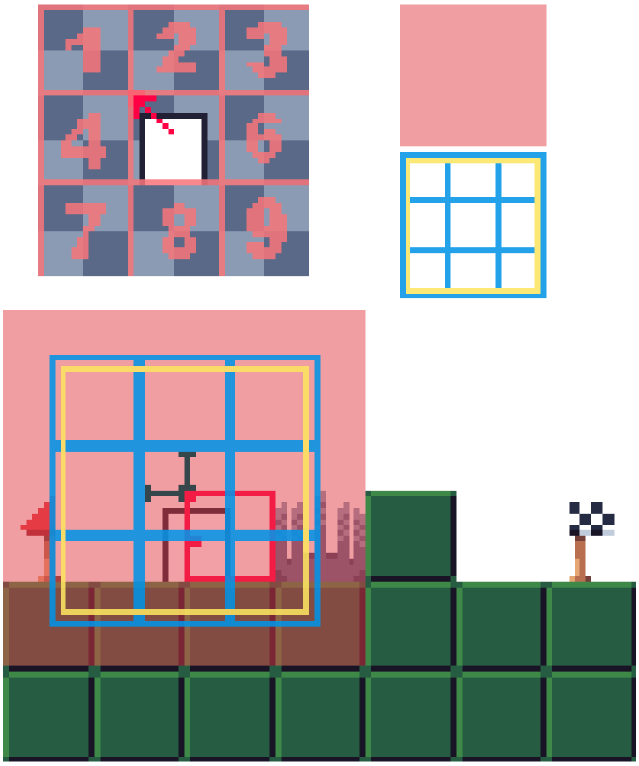
```
#
0.000000,0.098459
0.266460,0.365875
#
0.418636,0.554108
0.615794,0.823543
```

Figure 13 network file of the fittest network for the last generation

The resulting network connections can clearly be seen to increase in the correlation with the given target chromosome. Although this problem is trivial it is important to prove at this point that the underlying maths and theory works as intended.

At this point extra consideration had to accounted for, as these classes and function now had to be integrated into the game framework from the first stage.

It is important at this point to describe how the player would therefore receive input and in what form this would take.



The red box shows the tiles which are going to be taken as input by the user.

The blue boxes show the designated tile radius which is defined at a programmatic level so this would be 1 Up, 1 Down, 1 Left, 1 Right.
(This can be any range and will increase the number of tiles as input in a given direction if altered)

The yellow inner box is the rectangle which performs the tile interception check so any tile's sprite that falls under that box is counted as viewed tile.

Figure 14 Abstraction and visual overlay of the way the player and neural network will perceive the level and where it is.

If the player is to be moved by responding to the output of the controlling neural net then all that must be done is to hook up the output of the players corresponding neural network to the movement functions that are defined in the player class.

Therefore, the neural net needs to have a set of inputs to feedforwards from. A visual way to represent what the neural net can 'see' is shown in figure 14 above.

As each tile has an id that relates to what the type of tile it is in game. There can be a simple translation from a distinct list of tiles:

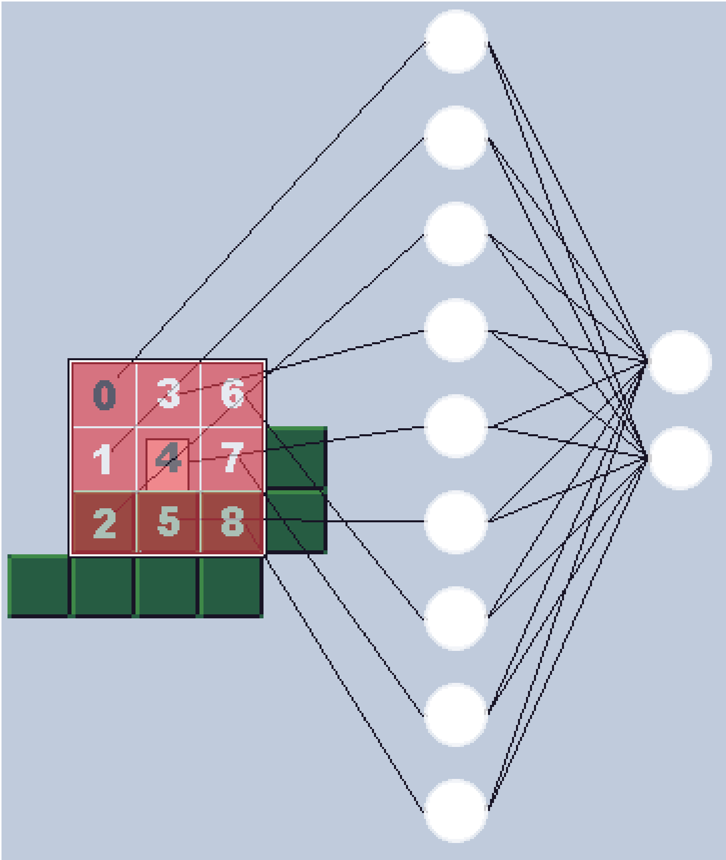
1. solid tiles (Ground)

2. non-solid tiles (Air and flags)
3. danger tiles (traps and level edges)

By attributing these tiles with different floating-point values. A list of the values corresponding to the tiles can be passed to the input of the neural net.

The code snippet for the view of the controller can be found at [10.1.10]

Shown in figure 14, The part in the top left shows how the first step is to get the centre of the player sprite and locate the top right pixel position of the tile that the player is in currently. This blue box is centred on this point and the yellow box is created from a set of scalar inputs for the distance to view in each direction: up, down, left and right. Starting with a centre box growing outward. So, given the parameters up = 3, down = 3, right = 3, left = 1. The number of tiles that the neural net will accept at any point is (up+down+1) x (right+left+1). In this case it will be 35 tiles. So, we can say that the network population that needs to be initialised will have an input layer of 35 nodes. This coincidentally is the configuration for the controllers view that made the most sense and is what is used in the training scenarios.



The output layer consists of 2 sigmoid nodes that link to the jump and move player functions. One is checked to see if the output ranges from (-0.9f < 0.0f < +0.9f) in this range the players movement is decided left if below -0.9f, stop if in between the two boundaries and right if greater than 0.9f.
Jump is similar – if the value is above 0.9f then call jump function else stop jumping.

Figure 15 Visual representation of the controller's view translated into the input of the network, example 3x3 view

At this point a population of configured neural networks can be generated and attached to the players so that they are controlled by their output. It is a good point to mention that an extension of the player class is was made to be able to do this and have the background to assign a fitness value to the player and control the state of the entity with more detail based on the new training state that encompasses this section of training. The outcomes and reasoning behind the way the networks are trained will be covered in more detail in the later section.

A large part of evolutionary algorithms is focused on the fitness function that is aliped to an individual for a certain problem.

To break the problem that is being presented here into its core foundations will assist in understanding how to calculate what fitness for an individual's performance.

The very simple way to describe the problem is ‘how far can an individual get to the right, without dying?’. At this point that is all the problem is. As the level design currently doesn’t account for paths that back track or require the player to plan 2 steps ahead. That would be much more problematic to address and would be something to consider for further research. However, the problem is very simple and requires a simple fitness function as a result. The way the fitness function is calculated for an individual’s performance on any given level is what percentage of the level did it complete (start to finish)?

Despite this being simple there are some key optimisations that can be used to improve the algorithm. If a player does reach the end then they should have a significant boost to their fitness over an individual whom completed 95% of the level. This is done so that during selection when parents are selected from the population pool the difference in fitness ratio between a 100% and 95% is much larger, therefore, favouring the individual whom completed the level significantly. This will push the population to prefer the networks that complete the levels entirely, while still diluting the population with diversity from the semi completed solutions.

This can be improved further (however was not full implemented and therefore not utilised in training) by introducing a time to complete for each player. An addition to the players fitness from how quickly an individual completed the level. This then favours those that complete the level faster than others and would be considered to have discovered a more optimal solution.

During the training cycle the player population is given a time to live, if the all the players have finished or died or the time to live the been exceeded then all the player are killed off and assigned their fitness for how far they got. Only once this has happened can the population be evaluated, and the next generation be created as shown in figure 11.

A check is also made on each player to see if they have made progress every so often, if not they presume that agent is stuck and kill it. This is just speed up the training process significantly as usually the first few generations are featureless and will just sit there.

To test this stage and make sure that the algorithm correctly functions some validation levels were created. These validation levels where designed to see if a solution could be found for a specific task/problem with into level. By making each level with a unique obstacle and still obtaining complete solutions the algorithm is functioning correctly. The list of validation levels can be found in the Supplementary materials filepath “Resources/level/validationlevels”. Each level has a different obstacle from the last, they are simple for a human who understands the controls to complete but the population will have to learn from scratch what its environment means and how to get as close to the end as possible in each situation. The levels consist of single pitfall, double pitfall, spikes/traps, stairs, and a combination of all of them. An example validation level is shown in figure 7 with a spike trap.

At this point just training on validation levels to prove that a single solution can be found is good. The main draw of this section is to see if by training on the randomly generated levels, the network controllers can traverse the validation levels and to see what features they develop and how well they generalise the solutions to translate them into solutions for the validation levels. By doing this it can be proven that the evolution of the generations is creating a suitable generalised solution to levels that the networks might come across. The training details and results can be found in the later section.

This is important to talk about in the development sate however, as this does add some requirements to the stage. Now there must be consideration made for a validation state that is created after the training state has completed it evolution runs. The state must test the population against the validation set to see the performance over all the individual levels after being trained on a separate level. Now this provides the data that can be analysed so that an understanding of how well the evolution is generalising the solutions.

State 3: Co-Evolution, increasing complexity

The aim of the co evolution stage of this project is to demonstrate how by evolving the population of player controllers and levels side by side, this can force both populations so find more complex features and behaviours. By evolving the complexity of the space and environment that the controllers are traversing it gives the evolution of the generations a *direction or heading.* This means that each population will try to improve what they are currently bad or not so good at. By doing this in conjunction with each population and having a fitness function that relies on the performance of one another, there is generated an oscillating pattern where by each side is forcing the other to get better, then to adjust and adapt to each other's advances.

In this context the fitness of a level is defined by what percentage of the player population can't complete the level. This include the edge case of if none of the players can complete the level then the fitness is greatly reduced. So, this will favour levels that the player population can still complete but finds hard to generate solutions to. The players fitness is altered slightly from the previous stage whereby the fitness of the player is given by the average percentage completed for each level in a generation. This should favour players that are better on average over the all the levels.

The design requirements for this stage are:

- Evolutionary and Genetic Algorithm should allow a population of objects that inherit from the IFitness interface

- Alter the functions currently in place to allow to type templating so that a population of levels and neural network-controlled players can use the same base code
- The IFitness interface includes the requirements for the child class to be used in the genetic algorithm
    - Chromosome, fitness and fitness ratio
    - This means refactoring to make the Level and NNControlledPlayer extend the IFitness interface
    - Also refactoring of previous training and validation states to accommodate change
- Independent mutation and crossover functions for a given object, Level and NNControlledPlayer.
    - This also allows for future classes to inherit the fitness interface and be implemented into a Genetic Algorithm with minimal effort.
- Co-evolution state

The refactoring of the other code stages can be seen by examining the neural folder in the code supplementary materials and the Level and NNControlledPlayer classes.

This stage introduces the need to be able to mutate and crossover two levels. This therefore requires a way of altering the levels in a safe way. Safe being a way to alter the levels so that the integrity of a 'level' is till maintained. This being there is a start and an end flag and checkpoints in the middle, while also making sure that the level is valid (completable).

The approach for relies on the ability to take a level and split it into its component sections that at all valid levels. So, a larger level with multiple checkpoints is one level with multiple sections that can all be converted into their own stand standalone level. By doing this and having access to an array of sub sections of a greater level opens the doors for changes and alterations to each section. The code for splitting a level can be found here [10.1.11]. This code snippet also is dependent on a levels ability to be converted into a chromosome and then to multidimensional array of sections or tile maps which correspond to each subsection. This is done so that the array of sub levels can be altered easily then converted back into a level after the changes have been made.

Details on the crossover and mutation function regarding the levels will be covered in the training section.

The co evolution state is essentially performing tournaments between both the player population and the level population. This is like the validation state where the player population is measured against a set of levels. Here a tournament matrix is used to determine the correlation between the two populations. Except the difference here is that this happens of the course of multiple generations with both populations evolving over time. The tournament matrix used here is a simple grayscale image. Whereby the rows denote the player population and the columns the level population. The colour that is found at the cross over point is an interpolated colour values based on the percentage completed for that specific level. A white completely white square shows that the player completed that level, then a gradient down to black for players that had no fitness or no activity. If you were to generate a luminosity histogram based on the image, then you would be able to see the frequency of the percentage completion at each generation and how it changes over time.

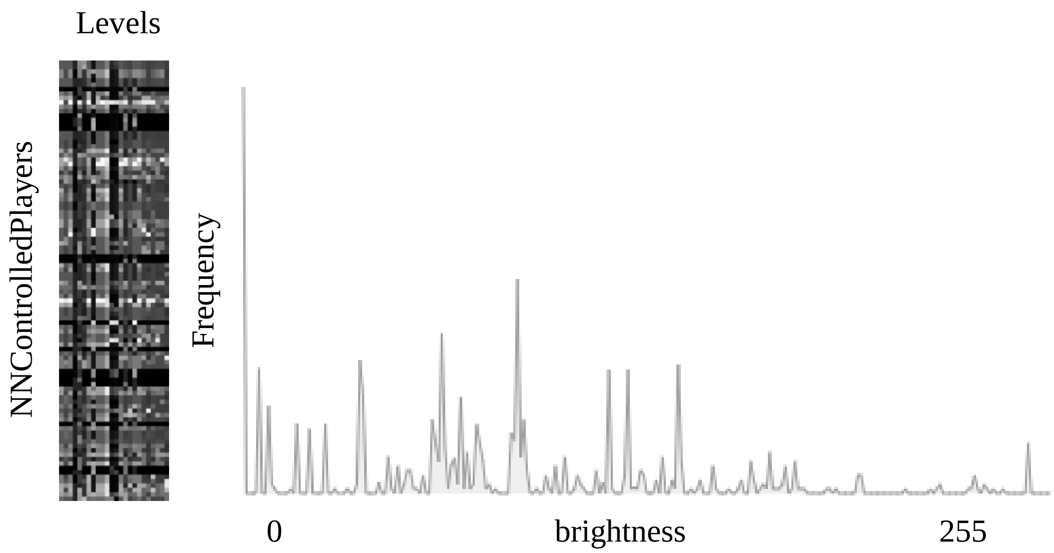


Figure 16 Tournament Matrix and histogram of generation 62 of a co evolution training run.

In figure 16, the histogram from this image shows that there are wide range of values occurring at a late generation. The height of the peaks denotes the frequency of a given value/colour. We can analyse this to find out what changes between generations.

6 Results

Neural Network trained by Evolutionary Genetic Algorithm

The training run in the TrainNetworkState and Validation State produce a folder that contains the last population of the trained network, training fitness data (csv), validation data (csv), tournament matrix (png).

In order to prove the base functionality of the evolutionary algorithm and the networks control capabilities, a simple training cycle was done on different typologies of network with different controller view sizes.

Side note, there will be fluctuations in the ranges of fitness functions that can be seen in the data, this is because some data that was collected from an earlier stage during development had different scaling for the fitness function. These results are still valid and are useful to evaluate.

The first set of data is from various populations on a few different training levels with differing topologies. This is using the training level 2 as the control.

For the purposes of all the training assume that the controllers view is configures as (3 up, 3 down, 1 left, 3 right) The explanation for this will be in the evaluation section. Also 120 is the default population size. If the generation number starts at non-zero then that just means that the fitness was always zero for the entire population, therefore it will be truncated.

The mutation rate for these experiments are set to 10% this is for both neural networks and the levels. As both have different internal workings it is hard to

compare the mutation rate. This is value that was found to produce the most constant results. In future works it would be prevalent to research the effect on the generation of the levels based on differing the mutation and crossover rates and population sizes.
The functions for Level mutation and crossover can be found in "Neural/GeneticAlgo.h" this header file contains the specialised functions for both the level and the neural networks. The mutation of a level can alters it by adding a new section, swapping to sub sections, shuffling the levels sub section or deleting an entire section. This then leads on to a subsection mutation where by the column in the section are changed. The cross over function has a 2% chance to occur whereby the sub sections of the parent level are mixed into to two off spring by using an altered version of the crossover point.
The neural network mutation will loop through every weight in the network and randomly add a change to some of them. This addition is based on a normal distribution centred on 0 with a standard deviation of 0.3f. The cross over function uses the same crossover point function but also has a chance to average the two parents chromosome which dramatically changes the networks structure.

6.1 Network training examples different topologies

6.1.1 35 : 2

Table 1 Training data for training level 2 - no hidden layers

Training Levels:	Resources/level/trainlevels/lvl-2		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	29.52605	1.587059	0
2	74.94322	10.73581	0
3	88.41435	19.39269	0
4	100	29.34694	1

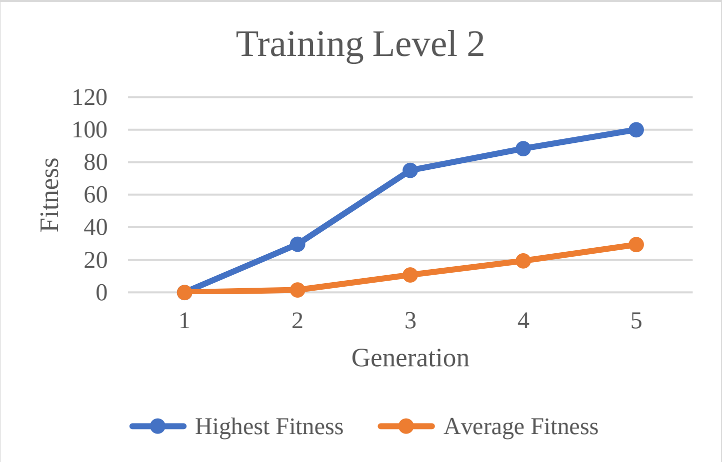


Figure 18 training level 2 graph - no hidden layer

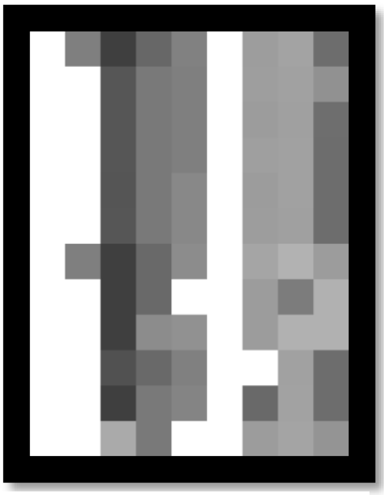


Figure 17 training level 2 no hidden layers tournament matrix on validation levels

6.1.2 35 : 6 : 2

Table 2 Training data for training level 2 - 1 hidden layer

Training Levels:	Resources/level/trainlevels/lvl-2		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	0	0	0
2	0	0	0
3	6.80042	0.05667	0
4	74.97046	5.437088	0
5	74.27245	12.44343	0
6	100	18.5077	1

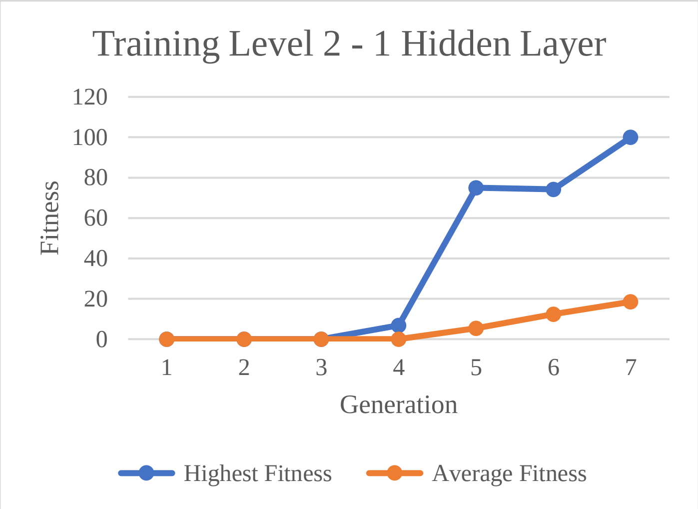


Figure 20 training level 2 graph - 1 hidden layer

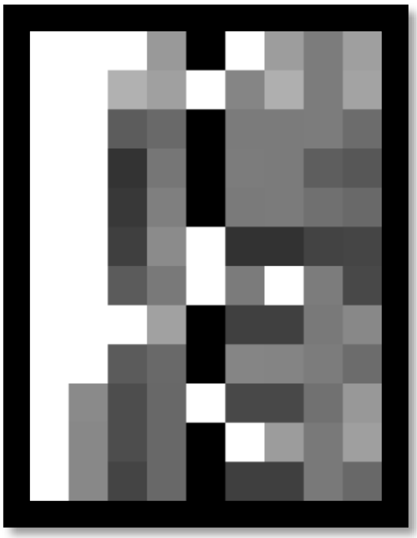


Figure 19 training level 2 - 1 hidden layers. tournament matrix on validation levels

6.1.3 35 : 6 : 6 : 2

Table 3 Training data for training level 2 - 3 hidden layers

Training Levels:	Resources/level/trainlevels/lvl-2		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
13	0	0	0
14	8.451505	0.070429	0
15	8.798384	0.750962	0
16	13.03518	3.511635	0
17	37.35502	4.582036	0
18	74.87587	5.799448	0
19	77.93457	6.064717	0
20	78.89014	7.286016	0
21	77.90221	12.90882	0
22	80.90731	15.2814	0
23	92.79741	19.74684	0
24	81.31243	18.39683	0
25	100	27.33634	1

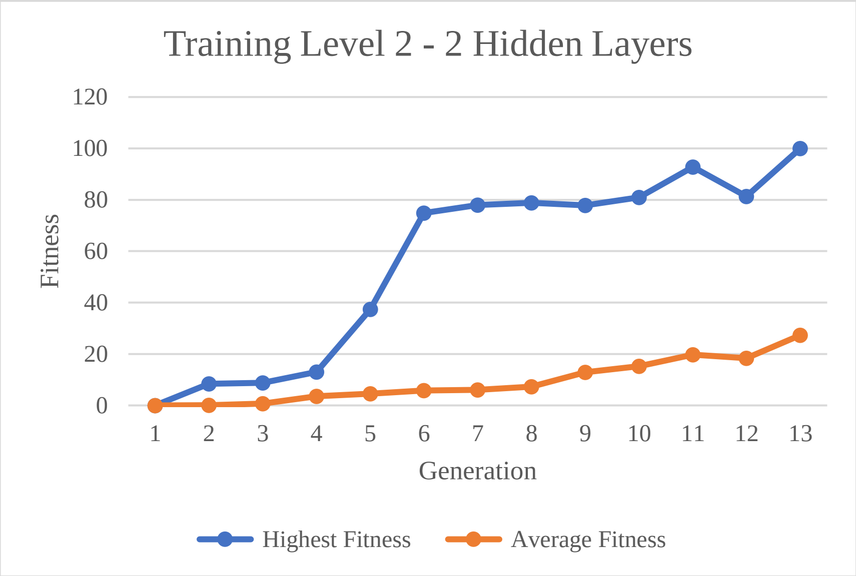


Figure 21 training level 2 graph -2 hidden layers

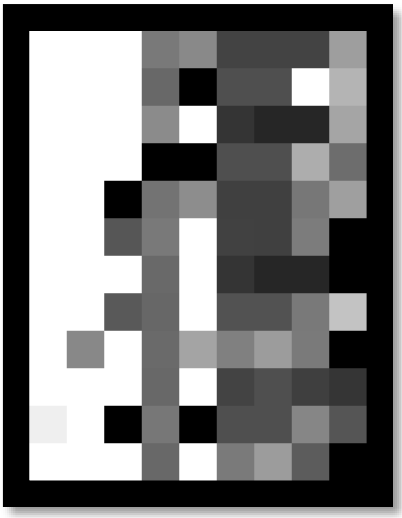


Figure 22 training level 2 - 2 hidden layers. tournament matrix on validation levels

6.1.4 35 : 10 : 6 : 6 : 2

Data Table [10.1.1]

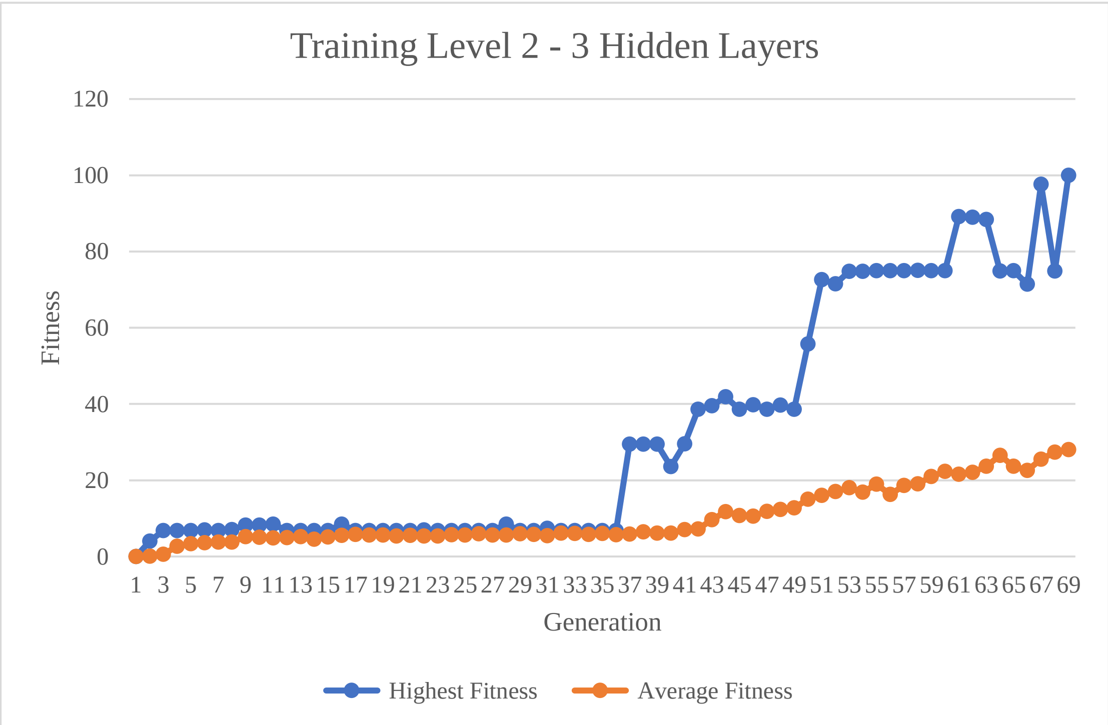


Figure 23 training level 2 graph - 3 hidden layer

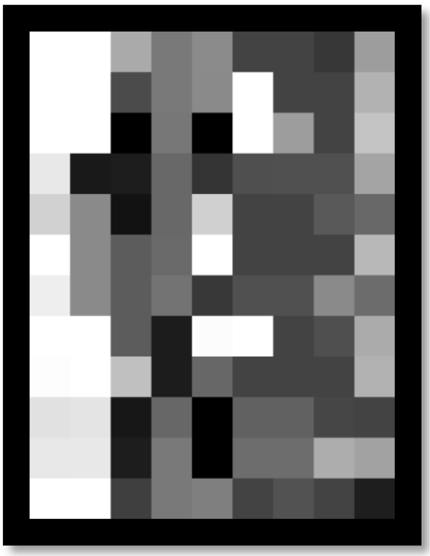


Figure 24 training level 2 - 3 hidden layers. tournament matrix on validation levels

6.2 Training Neural Network on randomly generated levels

Note: The rest of the training data for the training levels can be found in the supplementary materials "Resources/networks/training-level * validation" The graphs for them follow the same pattern as the other levels and are all completed properly.

6.2.1 Training level 0

Assume that the topology of the network population is 35 : 2

Data Table [10.1.2]

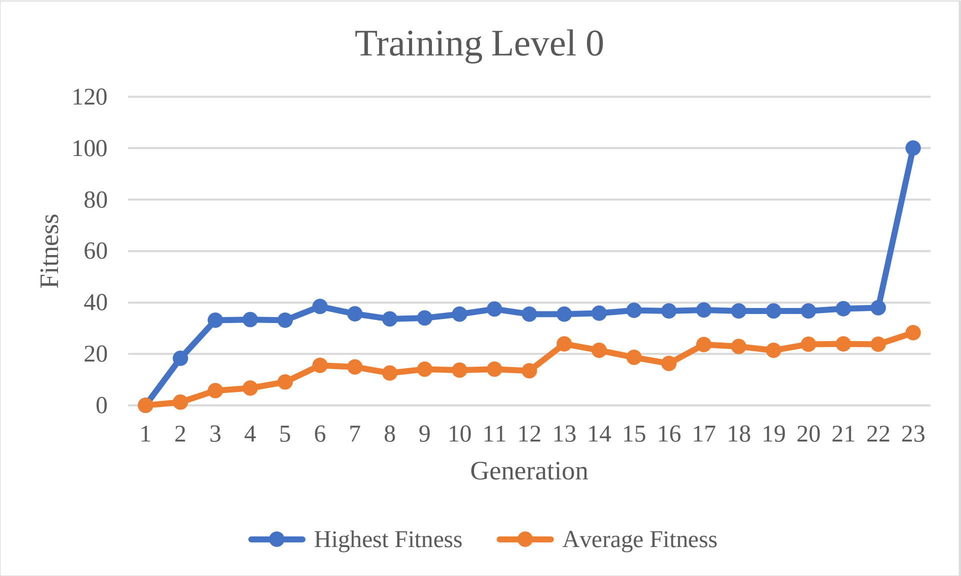


Figure 25 graph of training data for level 0

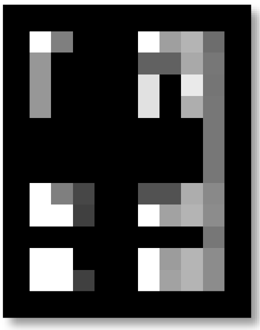


Figure 26 Matrix for the validation levels

6.2.2 Training level 1

Table 4 Table for training level 1data

Training Levels:	Resources/level/trainlevels/lvl-1		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	36.1352	1.405637	0
2	82.70406	10.20533	0
3	90.54327	18.86192	0
4	100	44.30504	4

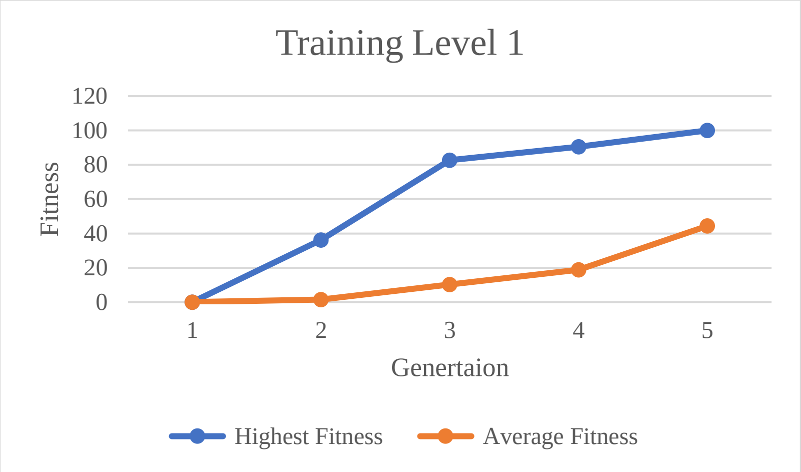


Figure 28 Graph of training level 1 fitness data

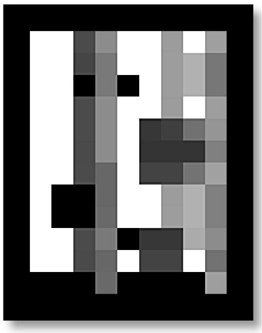


Figure 27 Validation Matrix results

6.2.3 Training level 3

Training for level to was covered earlier

Table 5 training data for level 3

Training Levels:	Resources/level/trainlevels/lvl-3		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	15.27283	2.28802	0
2	59.28617	7.490392	0
3	59.28617	8.588301	0
4	47.08181	8.193709	0
5	61.80167	12.07804	0
6	100	19.78092	1

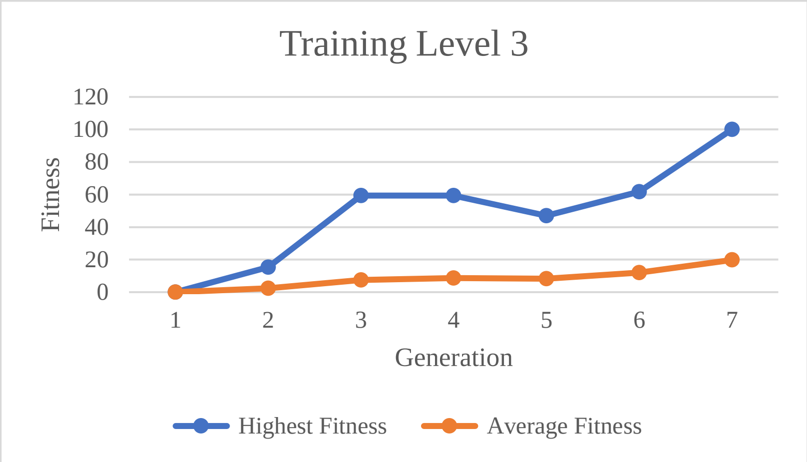


Figure 30 graph for training data level 3

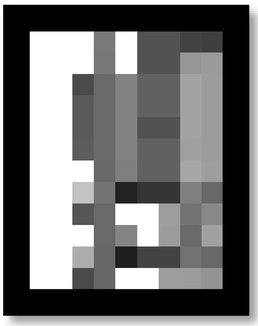


Figure 29 Validation Matrix

6.2.4 Training for level 4

The data table for level 4 [6.2.4]

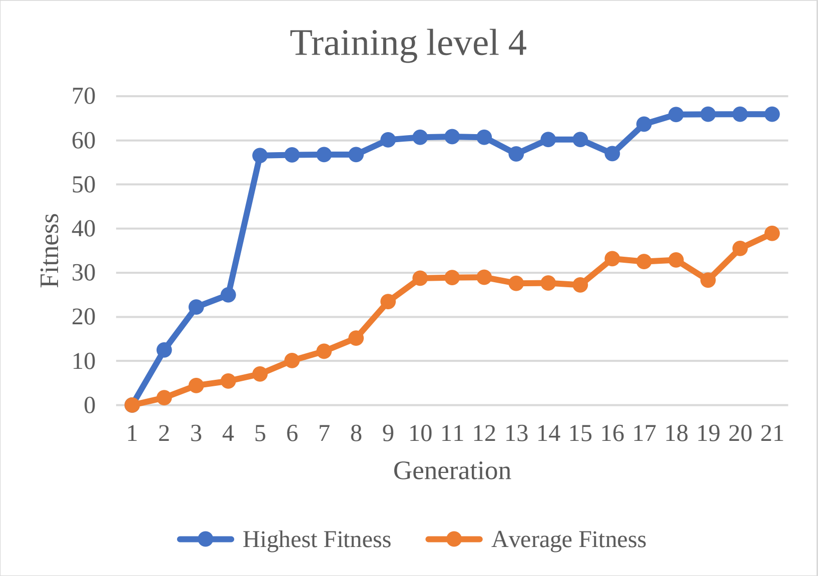


Figure 31 Graph for training level 4

There is no validation matrix for the level as it didn't find a solution the main level

This level didn't reach a solution in 100 generations. The level in question has a tricky placed spike trap and the network couldn't evolve to get over it properly. Despite the level being valid.

All the data for these training sets can be found in the supplementary materials "Resources/networks/training-level * validation"

6.2.5 Training levels 5-9 validation matrices

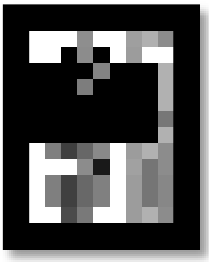


Figure 36 Validation training level 5

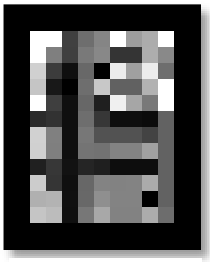


Figure 35 Validation training level 6

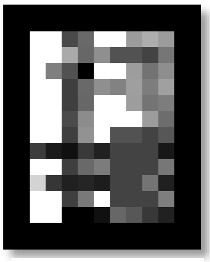


Figure 34 Validation training level 7

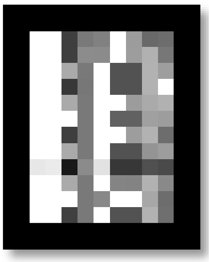


Figure 33 Validation training level 8

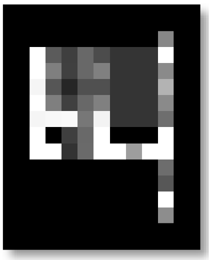


Figure 32 Validation training level 9

6.3 Training on Validation Levels and validating on training levels (swapped)

6.3.1 Validation Level 0

The first two levels are completed in one generation so no table or graph
All the data for these training sets can be found in the supplementary materials "Resources/networks/validation-level * training"

Figure 38 Validation Matrix for validation - level 0

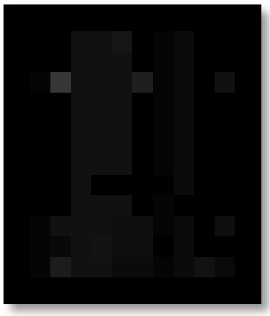


Figure 37 Validation Matrix for level 1

6.3.2 Validation level 2

Table 6 validation level 2 fitness data

Training Levels:	Resources/level/validationlevels/lvl-2		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	36.13275	6.52301	0
2	36.19785	16.18426	0
3	100	45.73138	5

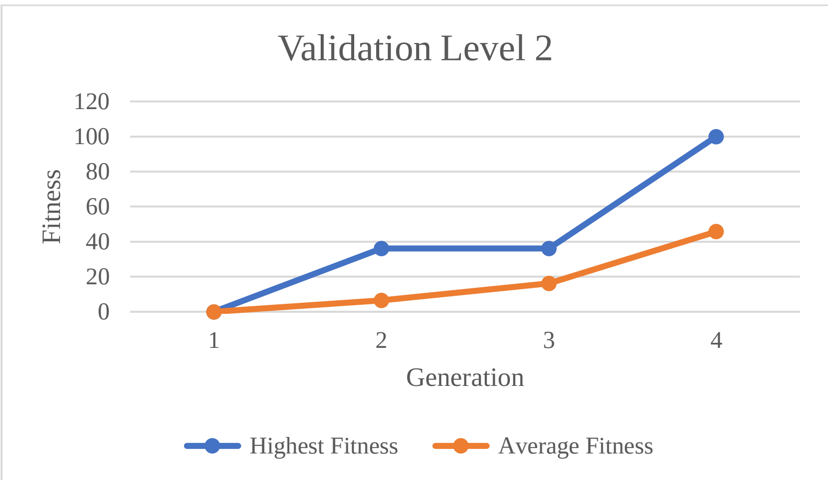


Figure 40 Graph for validation level 2 fitness data

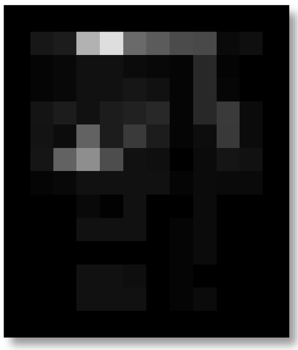


Figure 39 Validation Matrix validation Level 2

6.3.3 Validation level 3-8

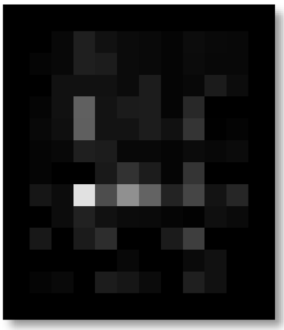


Figure 46 Validation Level 3

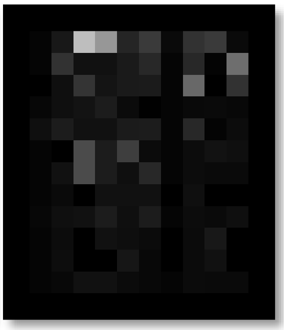


Figure 45 Validation Level 4

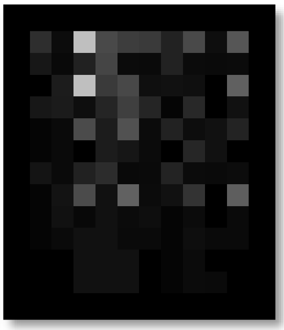


Figure 44 Validation Level 5

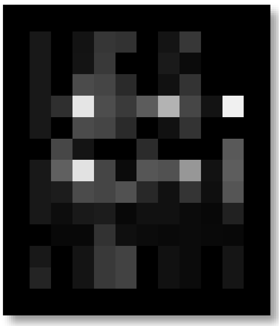


Figure 43 Validation Level 6

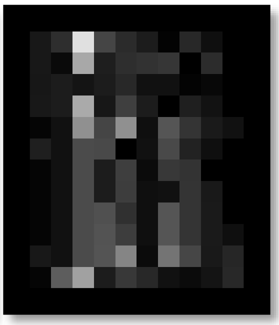


Figure 42 Validation Level 7

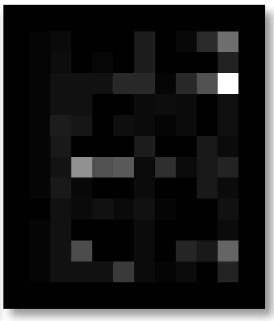


Figure 41 Validation Level 8

6.4 Co evolution of levels and player controllers

For these sets of data there is many validation matrices.

The for the purposes of these data sets the max number of generations is 100. The first data set was cut short due to a crash from a known bug which does not compromise the integrity of the data. Since this is essentially an open-ended task it seemed reasonable to halt at 100. If the program could run on a super cluster then the more generations wouldn't be a problem. When the level sixes get up in the section numbers the wait time is quite substantial.

6.4.1 Training set 1

Training data set Network and Level Population [10.1.4.1][10.1.4.2]

Graphs for the above data [10.1.4.3][10.1.4.4]

6.4.2 Training set 2

Training data set Network and Level Population [10.1.5.110.1.4.1][10.1.5.2]

Graphs for the above data [10.1.5.3][10.1.5.4]

7 Evaluation

From the results that have been collected from the co evolution test sets that have been run. It would be fair to say that over the course of 100 generations a level in the population will increase in difficulty. By looking at [10.1.4.3] the fitness graph generated by the level population, it can be determined that the levels are changing to become harder for the current population of controllers. The spikes in highest fitness line is when a level changes in a way that the current players can't complete it. After every spike there is negative gradient which correlates with the oscillating pattern of the player population. By having a steady oscillating line of the highest and average fitness of the player population this can be inferred that at this push and pull effect is creating harder levels and in turn the complexification of the space is improving the solutions found by the player population. This section of the project is a success. However, the approach taken to the whole project is not without scrutiny. In (Sorenson & Pasquier, 2010) it is stated that "current approaches follow a bottom-up, rule-based approach. This method requires a designer to embed aesthetic goals into a collection of rules, resulting in systems just as difficult to construct as hand-designed levels". It could be said that the approaches that have been taken in the design stages might have been short cuts to produce the desired result, essentially by constraining the generative algorithm to conform with a valid level there might be very little benefit from doing all this work, over old fashion level design. This is also a problem with the base framework of the game. If the levels had something else. For example, if other entities or coins where implemented into the generation stage of the project there might have been more interesting results. Perhaps if this project where to be undertaken again with this in mind, less time would be spent on the specifics of the genetic algorithm and more on implementing the features into the game that where the original aims of the project.

8 Conclusion

The three stages of development for this project outlined a framework where levels in a 2D platforming game could be generated through co evolution and neural networks. A game framework made from scratch using c++ and SFML proved to have tackled the toy problem that was created. At the end of this project it can now be realised that only the surface has been scratched in terms of the utilises of this framework and the approach of the level generation. The project could be taken in many different directions. The framework that has been built here has the potential for a lot more that has been utilised or demonstrated. For example, a short implementation of the neural network training could be used to implement a stylised level tutorial function, where by as a real player is figuring out a hard level, in the back ground the genetic algorithm is working on a solution and can be shown to the player in different section to provide hints on how to complete the level. This could be fleshed out into a greater game player mechanic. Another approach could be to increase to complexity of the base problem, right now the problem of solving a level by teaching a network to jump and go right at certain points is trivial, but by adding a layer of difficulty by requiring the need for a key to a door or having back tracking like in a metriod style game. This would then create the need for a more in depth approach to the neural network over having an encoder look at the tiles around the player, there would be the need for a decision making network that works in conjunction with movement network like the approach from (Robinson, 2007).

9 References

Gomila, L., 2018. *Documentation of SFML 2.5.1*. [Online]
Available at: https://www.sfml-dev.org/documentation/2.5.1/
[Accessed 31 03 2019].

Man, K. F., 1996. Genetic Algorithms: Concepts and Applications. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, 43(5), pp. 1-16.

Robinson, E. E. T. a. C. A., 2007. Neuroevolution of Agents Capable of Reactive and Deliberative Behaviours in Novel and Dynamic Environments. *European Conference on Artificial Life* , pp. 345-354.

Shaker, N., 2010. Level Generation Track. *The 2010 Mario AI Championship*, pp. 1-16.

Sorenson, N. & Pasquier, P., 2010. The Evolution of Fun: Automatic Level Design through Challenge Modeling. *ICCC*, pp. 258-267.

Yannakakis, G. N., 2011. Experience-Driven Procedural Content Generation. *IEEE TRANSACTIONS ON AFFECTIVE COMPUTING*, 2(3), pp. 147-161.

10 Appendices

Tables

10.1.1 Training Level 2 – 3 hidden layers

Table 7 Training data for training level 2 networks at 3 hidden layers

Training Levels:	Resources/level/trainlevels/lvl-2		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
15	0	0	0
16	3.997572	0.033313	0
17	6.80042	0.540981	0
18	6.80042	2.674499	0
19	6.80042	3.362725	0
20	6.929787	3.567471	0
21	6.80042	3.779672	0
22	7.060125	3.801748	0
23	8.236988	5.178529	0
24	8.236988	5.045278	0
25	8.451505	4.890456	0
26	6.817742	4.902862	0
27	6.80042	5.20494	0
28	6.80042	4.535549	0
29	6.80042	5.086851	0
30	8.451505	5.569549	0
31	6.817742	5.800936	0
32	6.817742	5.631875	0
33	6.80042	5.582124	0
34	6.80042	5.373665	0
35	6.80042	5.504885	0
36	6.981132	5.337999	0
37	6.80042	5.397931	0
38	6.80042	5.702701	0
39	6.80042	5.614238	0
40	6.80042	5.933172	0
41	6.80042	5.61529	0
42	8.451505	5.570152	0
43	6.80042	5.959231	0
44	6.80042	5.798871	0
45	7.368622	5.435637	0
46	6.80042	6.091377	0
47	6.80042	6.009754	0
48	6.80042	5.815696	0
49	6.80042	6.010099	0
50	6.80042	5.670079	0
51	29.48821	5.845742	0
52	29.48821	6.418338	0

53	29.48821	6.097185	0
54	23.60981	6.077153	0
55	29.52774	7.074485	0
56	38.61864	7.189339	0
57	39.54381	9.628368	0
58	41.89455	11.75679	0
59	38.61259	10.72215	0
60	39.77463	10.56953	0
61	38.62274	11.85211	0
62	39.73303	12.31366	0
63	38.62274	12.71293	0
64	55.71511	15.03307	0
65	72.59756	15.98157	0
66	71.51402	17.06949	0
67	74.79792	18.00695	0
68	74.8112	16.83497	0
69	74.95464	18.91958	0
70	74.97833	16.23547	0
71	74.95584	18.65752	0
72	74.99999	19.04897	0
73	74.94653	20.93359	0
74	74.94503	22.34556	0
75	89.15499	21.53855	0
76	89.01453	22.05867	0
77	88.41647	23.69426	0
78	74.9082	26.48582	0
79	74.97639	23.66443	0
80	71.46211	22.58426	0
81	97.66608	25.53303	0
82	74.90812	27.32394	0
83	100	28.04551	1

10.1.2 Training data for Level 0 – 35 : 2

Training Levels:	Resources/level/trainlevels/lvl-0		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	18.25272	1.258111	0
2	33.06108	5.678827	0
3	33.2919	6.680108	0
4	33.06108	9.145173	0
5	38.43203	15.56508	0
6	35.61715	14.91571	0
7	33.61151	12.62948	0
8	34.02691	14.1162	0
9	35.52912	13.7002	0

10	37.446	14.01375	0
11	35.52912	13.46827	0
12	35.52912	23.93932	0
13	35.84872	21.36702	0
14	37.01084	18.69292	0
15	36.68127	16.27075	0
16	37.07904	23.66672	0
17	36.70588	22.89687	0
18	36.70588	21.39612	0
19	36.71413	23.73685	0
20	37.62784	23.95428	0
21	37.9335	23.72465	0
22	100	28.21863	1

10.1.3 Training data for level 4 – 35:2

Table 8 Training data for level 4, as can be seen in the table there was no solution found the run continued to 100 generations while stuck in this local maximum

Training Levels:	Resources/level/trainlevels/lvl-4		
Gen Num	Highest Fitness	Average Fitness	Population Size: 120
0	0	0	0
1	12.46449	1.649185	0
2	22.19713	4.432615	0
3	24.96739	5.48441	0
4	56.57553	7.0848	0
5	56.68207	10.1179	0
6	56.76491	12.19775	0
7	56.8089	15.18574	0
8	60.08487	23.42575	0
9	60.65778	28.778	0
10	60.87085	28.89636	0
11	60.65778	28.97139	0
12	56.94902	27.60045	0
13	60.21966	27.63464	0
14	60.21966	27.21586	0
15	57.01351	33.19767	0
16	63.69168	32.54648	0
17	65.87357	32.86353	0
18	65.90261	28.32289	0
19	65.89133	35.487	0
20	65.89133	38.89942	0

10.1.4 Co-evolution training set 1

10.1.4.1 Network Population

Table 9 Training set 1 network population co evolution

Gen Num	Highest Fitness	Average Fitness
0	0	0
1	49.29282	11.48051
2	65.64803	22.20736
3	69.34343	29.09052
4	75.75555	33.40256
5	79.66589	38.51437
6	83.04547	36.17204
7	88.65807	30.74084
8	86.02126	33.78772
9	76.31567	37.09866
10	52.05783	21.85378
11	50.47263	10.63686
12	38.77313	10.76873
13	35.77836	11.65291
14	37.82627	20.523
15	31.95502	17.00351
16	34.18481	19.84444
17	46.00479	24.55598
18	43.65038	22.60803
19	36.12076	17.13737
20	37.60529	20.88554
21	38.9354	23.92636
22	36.82853	21.1877
23	50.09322	19.78472
24	70.32627	23.67187
25	84.46937	28.03072
26	61.45282	26.78908
27	47.56181	18.74941
28	32.81641	17.19226
29	57.27795	18.9479
30	62.55236	21.13772
31	57.05128	25.12477
32	53.08358	22.66282
33	49.89351	19.91094
34	47.33968	22.07145
35	42.71654	21.00084
36	35.83537	18.43479
37	34.55295	18.26888
38	36.22218	13.78866
39	34.71564	15.85811
40	39.86523	20.04195

41	36.75531	19.15615
42	26.7181	9.427524
43	42.28838	9.957682
44	35.15707	11.81356
45	36.61568	17.1661
46	54.19804	18.73754
47	56.86065	20.81144
48	54.73465	22.5399
49	66.55584	23.94915
50	65.70356	24.95229
51	68.26279	28.47253
52	70.6726	27.58798
53	61.18763	28.38752
54	68.11808	33.2019
55	66.84544	30.75103
56	72.01667	39.8073
57	71.68636	41.52895
58	68.03953	38.11351
59	70.60044	40.26104
60	74.11847	41.38466
61	59.38299	40.37265
62	54.24434	38.2412
63	60.38346	37.04556
64	47.93539	28.21387
65	34.99055	19.41596
66	47.27304	14.98335
67	37.95705	9.926817
68	28.56544	12.383
69	29.86909	12.2709
70	21.06523	10.60449
71	16.01893	8.656672
72	11.48248	6.651396
73	51.27776	7.370184
74	72.81289	9.487775
75	91.99081	16.34764
76	89.38929	24.81501
77	88.4491	32.58016
78	85.61215	35.02351
79	92.24505	39.28665
80	89.72327	40.57573
81	93.12024	40.7809
82	91.84668	45.48917
83	89.77454	46.81123
84	88.94712	34.2475
85	95.38113	25.77316
86	94.7829	35.84354

87	96.39416	38.51746
88	96.84232	34.61628
89	95.29691	36.35659

10.1.4.2 Level Population

Table 10 training set 1 data Level population co evolution

Gen Num	Highest Fitness	Average Fitness
0	0	0
1	83	62.45833
2	32	29.08333
3	25	18.58333
4	25	17.625
5	19	12.91667
6	73	11.91667
7	14	6.75
8	52	11.45833
9	55	15.66667
10	69	37.5
11	59	45.875
12	55	38.20833
13	76	34.04167
14	35	16.125
15	35	11.25
16	34	14.125
17	20	9
18	27	16.29167
19	22	15.29167
20	16	12.125
21	8	6.083333
22	25	9.208333
23	15	8.875
24	8	5.625
25	53	7.583333
26	52	3.958333
27	15	9.625
28	5	3.666667
29	4	2.625
30	5	4.375
31	1	0.958333
32	2	1.75
33	3	2.708333
34	3	1.166667

35	4	3.708333
36	4	3.666667
37	24	3.791667
38	3	1.416667
39	3	1.125
40	2	1
41	1	0.208333
42	84	3.666667
43	22	4.75
44	13	10.875
45	8	6.875
46	7	6.125
47	7	5.75
48	4	3.791667
49	9	4.583333
50	10	4.5
51	9	8.041667
52	6	5.833333
53	7	5.125
54	6	5.166667
55	10	8.625
56	7	4.791667
57	42	10
58	12	10.70833
59	9	6.416667
60	11	9.25
61	7	3.75
62	8	7.333333
63	82	10.625
64	82	20.04167
65	77	44.45833
66	78	58.95833
67	60	59.125
68	12	11.08333
69	14	13.16667
70	8	7.041667
71	6	5.041667
72	7	6.958333
73	6	5.958333
74	9	8.291667
75	9	8.791667
76	12	11.45833
77	6	3.583333

78	6	6
79	3	2.916667
80	11	6.208333
81	10	4.333333
82	4	3.166667
83	15	4.041667
84	16	7.458333
85	18	14.375
86	11	9.666667
87	7	6.958333
88	7	6.208333
89	5	3.166667

10.1.4.3 Level Population fitness graph

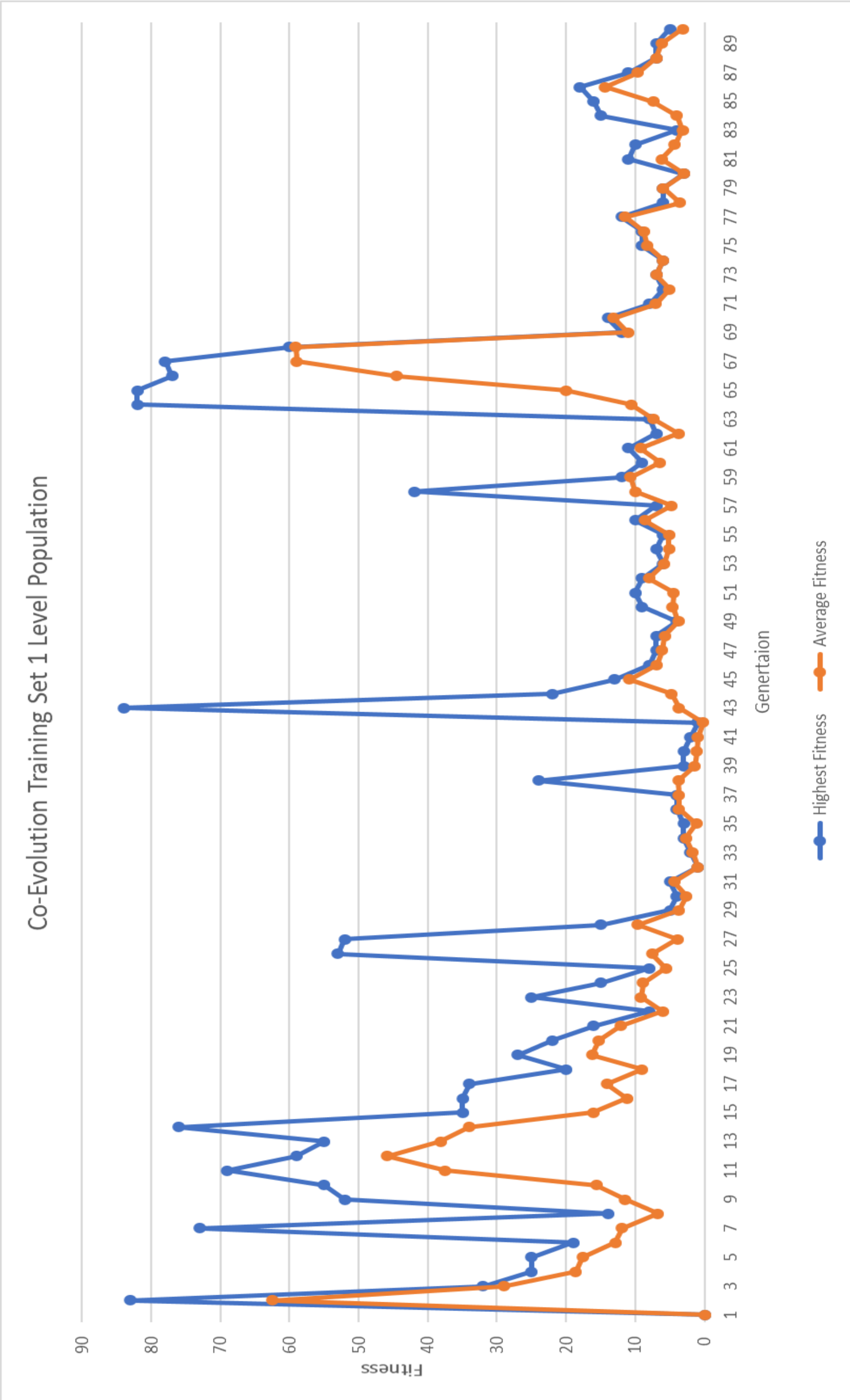


Figure 47 co evolution training set 1 Level Population

10.1.4.4 Player Population fitness Graph

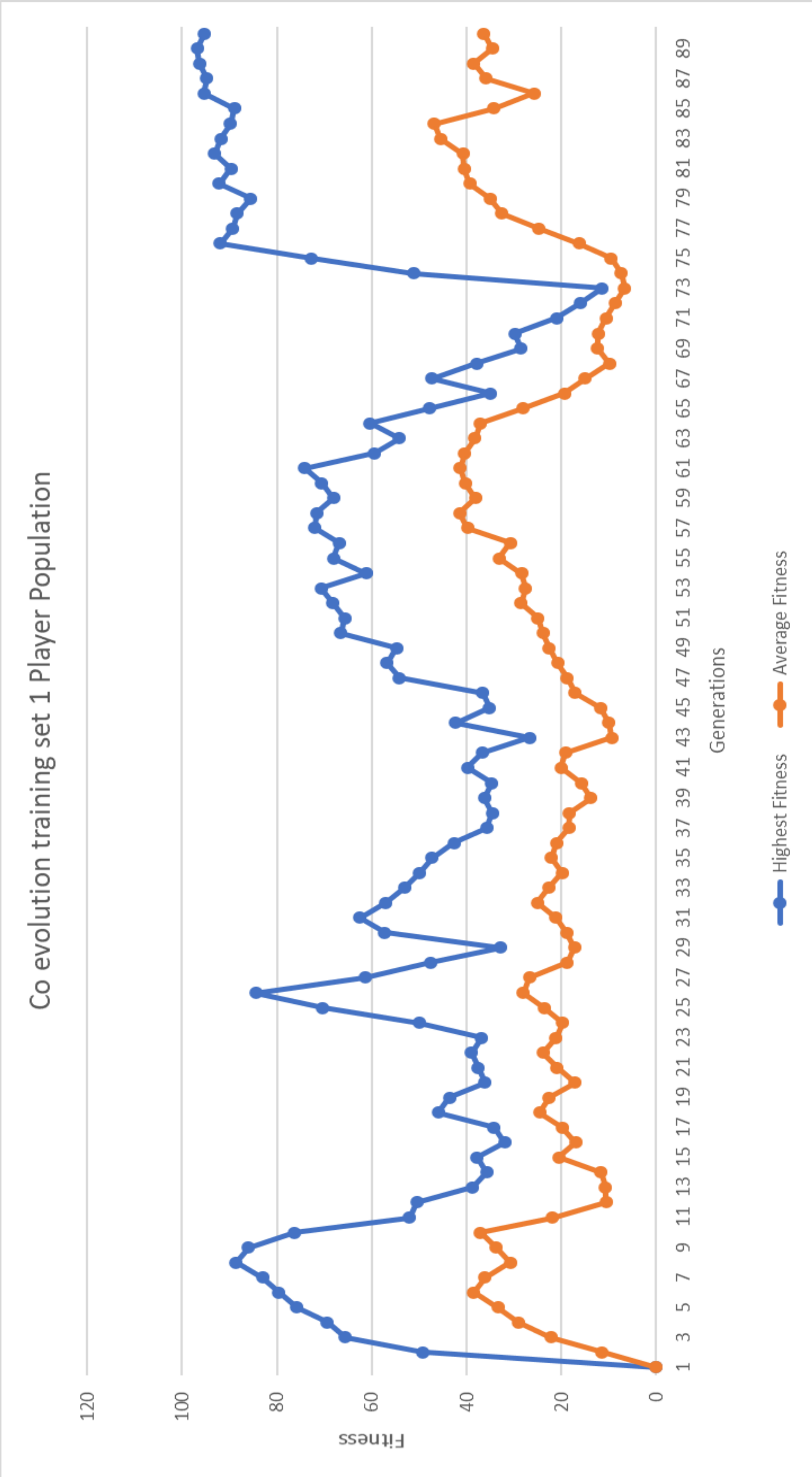


Figure 48 co evolution training set 1 Player population

10.1.5 Training set 2

10.1.5.1 Network Population Fitness data

Table 11 Network Population fitness training set 2

Gen Num	Highest Fitness	Average Fitness
0	0	0
1	58.73138	10.0163
2	70.96407	32.40334
3	75.73956	32.72841
4	90.94864	40.35025
5	90.61593	37.22803
6	79.16666	35.33526
7	57.75757	27.46382
8	32.68193	18.6456
9	26.82979	11.10956
10	25.92656	12.22654
11	71.13742	19.90029
12	69.25987	22.36326
13	62.89494	22.55955
14	48.60709	25.49694
15	39.98761	19.11135
16	28.11665	12.04887
17	21.79124	10.45671
18	14.14158	8.002066
19	20.25139	10.9499
20	24.95693	9.150263
21	41.89636	13.40132
22	46.45492	12.86209
23	33.19355	13.45087
24	31.01383	8.916373
25	26.31347	8.527494
26	21.60562	6.182147
27	44.86523	9.294208
28	45.52946	11.77519
29	69.00953	13.21852
30	55.86774	12.81656
31	45.62656	13.52107
32	70.58535	13.70127
33	35.44413	11.3454
34	71.23785	17.31067
35	80.12513	19.83847
36	44.99913	16.16455
37	43.2041	11.56289
38	28.04709	11.19764
39	40.1354	13.57498
40	44.60403	16.06409

41	37.73628	17.51901
42	57.12297	15.30712
43	36.41166	9.700733
44	38.72346	10.37729
45	38.61248	12.34629
46	34.03667	10.64015
47	33.78045	10.84889
48	37.54623	10.02366
49	41.44379	10.40925
50	27.06259	11.39771
51	25.95546	11.74904
52	27.0632	10.83191
53	26.32162	11.89057
54	33.29166	10.23067
55	30.76898	11.04422
56	39.30845	12.14423
57	37.9435	14.69735
58	58.21896	24.36439
59	69.25685	30.91022
60	65.43386	29.99184
61	71.37874	30.66449
62	70.29916	31.34321
63	61.7315	31.69692
64	58.7463	30.57289
65	55.94403	28.11263
66	70.21434	24.64254
67	27.20839	14.05849
68	28.36044	6.7164
69	27.39949	6.70616
70	54.23652	11.52539
71	36.73553	13.58394
72	37.68997	13.74859
73	53.43283	15.10976
74	43.0262	14.32502
75	49.1791	15.25069
76	28.18073	13.03029
77	29.28264	11.34951
78	34.07888	13.81558
79	42.97053	17.9077
80	35.57941	17.81623
81	50.26574	22.06396
82	54.6828	23.84107
83	39.81112	20.84209
84	43.59532	18.98724
85	35.65937	11.04426
86	36.56184	17.32263

87	40.65644	20.08273
88	38.2975	20.27061
89	37.08901	18.88258
90	50.11692	23.0936
91	39.13419	18.46901
92	48.66839	14.91322
93	33.56175	12.43144
94	28.7227	11.78428
95	24.23971	12.60869
96	34.01118	13.93688
97	37.69152	15.26957
98	19.91149	9.577997
99	26.84636	14.12668

10.1.5.2 Level Training data set 2

Table 12 Level Population Training data set 2

Gen Num	Highest Fitness	Average Fitness
0	0	0
1	73	70.375
2	58	25.70833
3	42	16.04167
4	15	12.83333
5	38	13.83333
6	38	14.79167
7	49	22.70833
8	49	32.20833
9	48	40.54167
10	33	31.20833
11	8	4.041667
12	5	3.75
13	49	8.916667
14	8	4.541667
15	73	8.666667
16	17	5.625
17	12	7.166667
18	15	12.75
19	14	3.833333
20	34	6
21	10	4.416667
22	33	9
23	14	2.958333
24	24	15.5
25	16	6.875
26	13	11.16667
27	5	4.458333

28	8	7.541667
29	2	2
30	5	1.916667
31	8	5.958333
32	55	7.708333
33	3	2.375
34	4	3.958333
35	1	1
36	89	10.58333
37	7	3.583333
38	9	5
39	6	3.708333
40	3	1.833333
41	14	2.791667
42	15	10.04167
43	23	16.875
44	9	8.75
45	4	3.791667
46	6	4.333333
47	8	4.5
48	15	10.25
49	7	5.625
50	7	5.625
51	3	2.958333
52	5	2.916667
53	5	4.75
54	2	1.958333
55	4	2.708333
56	17	2.416667
57	21	4.125
58	18	17.29167
59	1	0.958333
60	30	4.916667
61	27	5.333333
62	16	11.66667
63	11	10.625
64	2	2
65	6	0.25
66	2	1.5
67	41	19.625
68	46	38.70833
69	30	29.375
70	4	4
71	10	9.833333
72	1	1
73	4	3.666667

74	6	5.416667
75	19	4.625
76	28	10.04167
77	31	14.33333
78	28	21.25
79	5	5
80	3	3
81	0	0
82	5	1.166667
83	12	4.25
84	12	5.166667
85	9	7.125
86	7	6.416667
87	8	2.708333
88	4	3.166667
89	5	4
90	4	2.625
91	4	3.083333
92	4	1.875
93	3	2.833333
94	3	1.083333
95	2	0.958333
96	2	1.083333
97	42	1.75
98	3	1.333333
99	5	2.083333

10.1.5.3 Player Population fitness graph

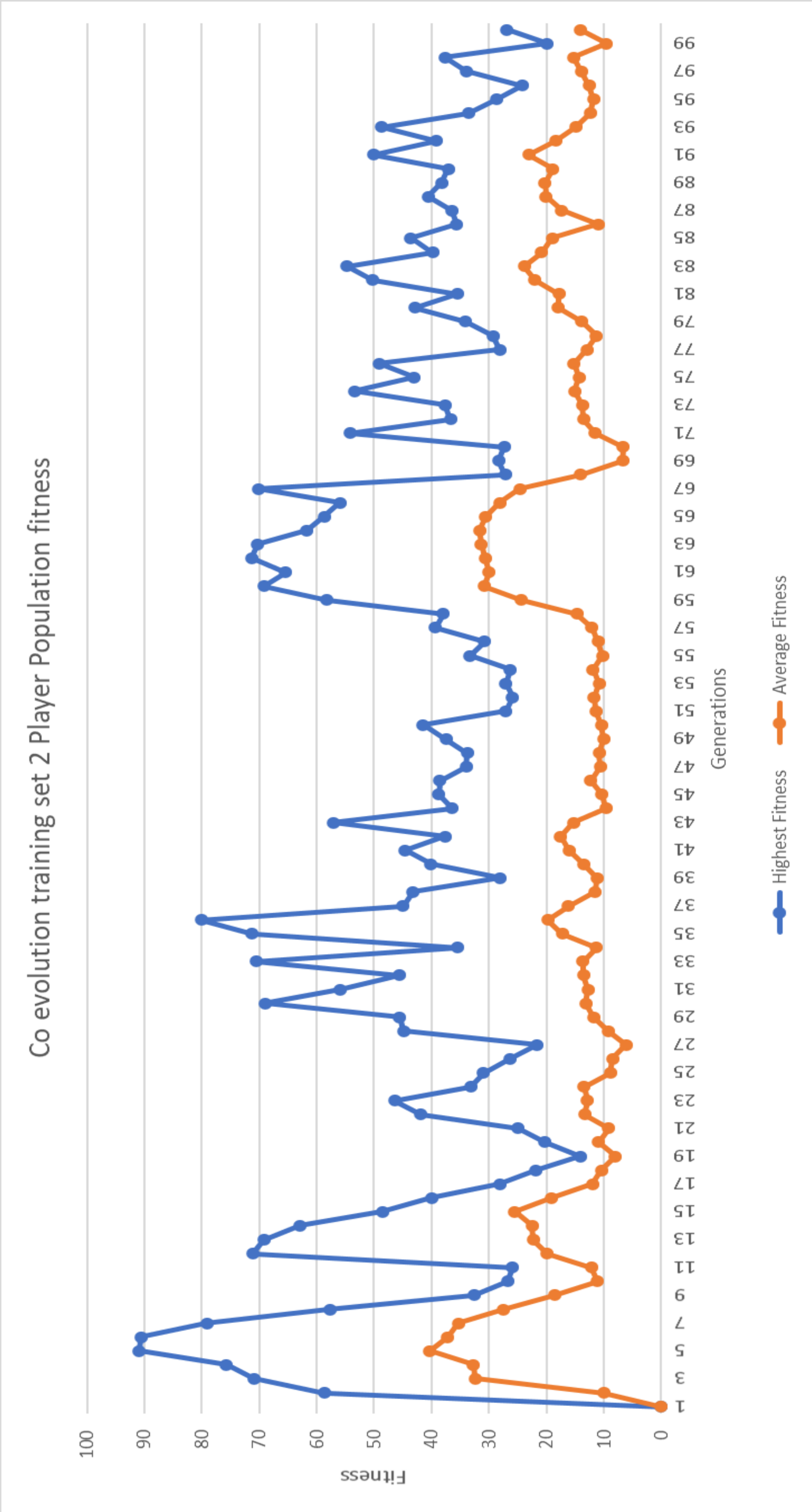


Figure 49 Player Population fitness graph for training set 2

10.1.5.4 Level Population fitness Graph

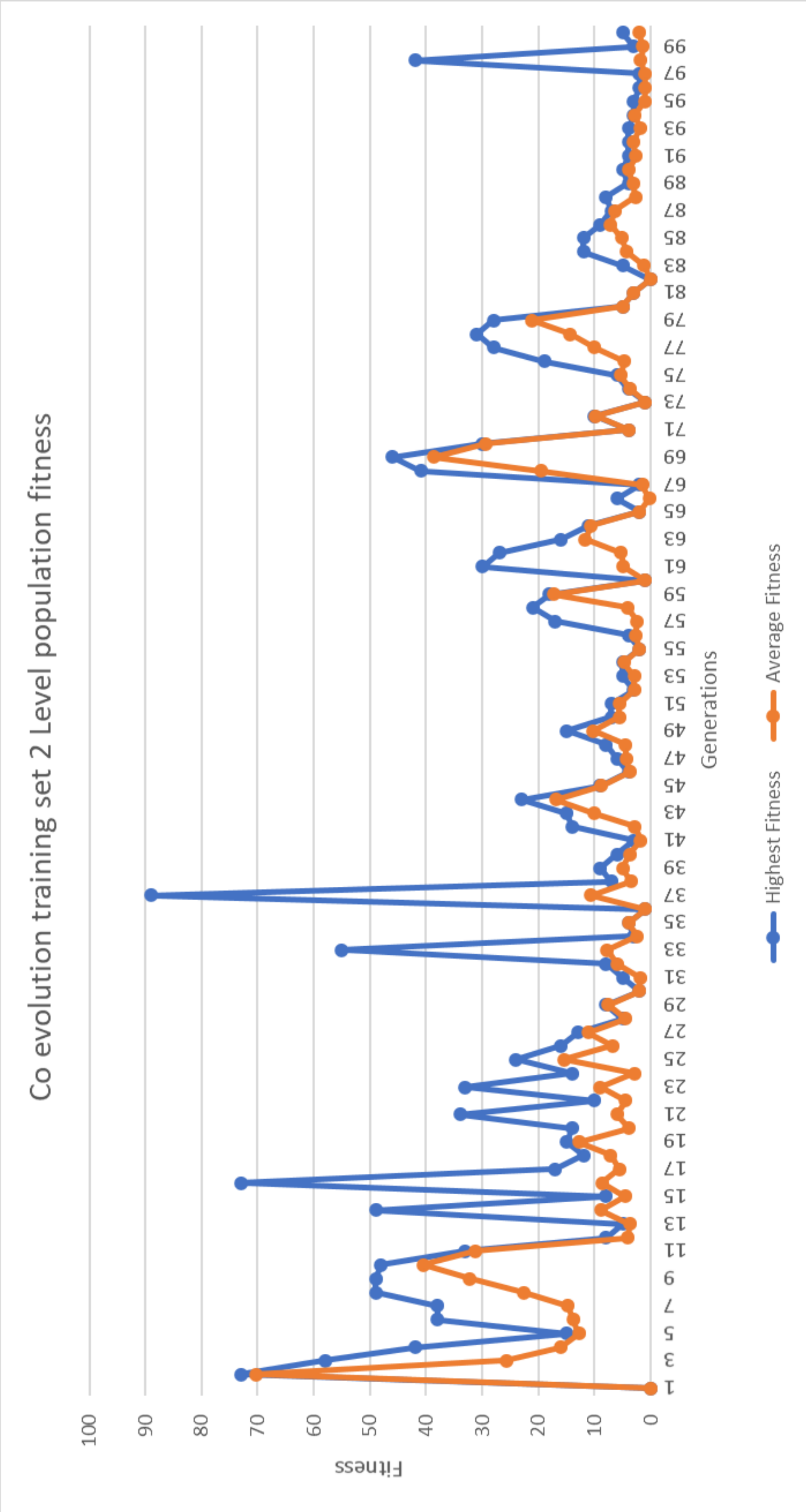


Figure 50 Level Population fitness graph training set 2 co evolution

Code Snippets

10.1.6 Game::run()

```
void Game::run()
{
	float newTime, frameTime, interpolation;
	float currentTime = this->_clock.getElapsedTime().asSeconds();
	float accumulator = 0.0f;

	while (this->_data->window.isOpen()) {
		this->_data->stateMachine.processStateChanges();
		newTime = this->_clock.getElapsedTime().asSeconds();

		frameTime = newTime - currentTime;
		if (frameTime > 0.25f) {
			frameTime = 0.25f;
		}
		currentTime = newTime;

		this->_data->stateMachine.getAvtiveState()->handleEvents();
		if (!this->_data->releaseAccumulator) {
			accumulator += frameTime;
			while (accumulator >= dt) {
				this->_data->stateMachine.getAvtiveState()->update(dt);
				accumulator -= dt;
			}
		}
		else
		{
			accumulator = 0;
			this->_data->stateMachine.getAvtiveState()->update(dt);
		}
		interpolation = accumulator / dt;
		this->_data->stateMachine.getAvtiveState()->draw(interpolation);
	}
}
```

Code Snippet 1 Game Run function

10.1.7 GameObjectManager::collisionCheck()

```
template<class T>
inline std::vector<T*> GameObjectManager::collisionCheck(sf::FloatRect hitBox,
ObjectLayer layer)
{
	std::vector<T*> entitesInArea = std::vector<T*>();
	T* eptr = nullptr;
	for (IEntity* e : this->_entities.at(layer)) {
		if ((eptr = dynamic_cast<T*>(e)) != NULL) {
			if (hitBox.intersects(e->getSprite().getGlobalBounds())) {
				entitesInArea.push_back(eptr);
			}
		}

	}
	return entitesInArea;
}
```

Code Snippet 2: Game Object Manager interlayer collision checking. This is done by checking to see if the given rectangle is intersecting with any entities on the layer that is specified. Returning an array of those entities that fall under that rect. This is only utilised in the game state to check for collisions with coins in another layer. However, this is a vital function if the game were to grow in the gameplay area. For example, there could be chests or keys that the player must collect or interact with. This function would allow for this functionality to be implemented easily.

10.1.8 Level::stichLevels()

```
void Level::stichLevels(Level & lvlA, Level & lvlB)
{
	this->_width = lvlA.getWidth() + lvlB.getWidth() - 3; //accounting for the
stiching process.
	Tilemap& tilemapA = lvlA.getTileMap();
	Tilemap& tilemapB = lvlB.getTileMap();

	int yposA = 0;
	for (int y = 0; y < int(lvlA.getHeight()); y++) {
		Tile& tile = tilemapA.at(y*lvlA.getWidth() + (lvlA.getWidth() - 2));
		if (tile.getTileID() == FINISH_LINE_TILE) {
			yposA = y;
			break;
		}
	}
	int yposB = 0;
	for (int y = 0; y < int(lvlB.getHeight()); y++) {
		Tile& tile = tilemapB.at(y*lvlB.getWidth() + 1);
		if (tile.getTileID() == CHECKPOINT_TILE) {
			yposB = y;
			break;
		}
	}

	int flagYPosDelta = yposA - yposB;//if its negative then add air to that
number of air rows to the top
	bool partAShift = false;
```

```
int newHeight = 0;
	if (flagYPosDelta > 0) {
		//shift lvlb down by that amount
		newHeight = lvlB.getHeight() + flagYPosDelta;
		this->_height = std::max(newHeight, lvlA.getHeight());
	}
	else {
		partAShift = true;
		flagYPosDelta = std::abs(flagYPosDelta);
		newHeight = lvlA.getHeight() + flagYPosDelta;
		this->_height = std::max(newHeight, lvlB.getHeight());
	}

	//given these two levels can we put them together to make a bigger one

	std::vector<std::string> tileData = std::vector<std::string>(this-
>_height*this->_width);

	for (std::string& td : tileData) {
		td = "61";
	}
	int index = 0;
	std::string tileString = "";
	for (int x = 0; x < lvlA.getWidth()-1; x++) {
		for (int y = 0; y < lvlA.getHeight(); y++) {
			index = y * lvlA.getWidth() + x;
			Tile& tile = tilemapA.at(index);
			tileString = std::to_string(tile.getTileID());
			tileString = (tile.getTileID() < 10) ? "0" + tileString :
tileString;
			if (partAShift) {
				tileData.at((flagYPosDelta + y)*this->_width + x) =
tileString;
			}
			else {
				tileData.at(y*this->_width + x) = tileString;
			}
		}
	}
	for (int x = 1; x < lvlB.getWidth(); x++) {
		for (int y = 0; y < lvlB.getHeight(); y++) {
			index = y * lvlB.getWidth() + x;
			Tile& tile = tilemapB.at(index);
			tileString = std::to_string(tile.getTileID());
			tileString = (tile.getTileID() < 10) ? "0" + tileString :
tileString;
			if (partAShift) {
				tileData.at((y*this->_width) + (x + lvlA.getWidth() -
3)) = tileString;
			}
			else {
				tileData.at(((flagYPosDelta + y)* this->_width) + (x +
lvlA.getWidth() - 3)) = tileString;
			}
		}
	}
	writeTileData(tileData);
}
```

Code Snippet 3 Level Stitching function, this is used to combine two levels with given tilemaps to make a larger level.

10.1.9 Player::update() < collision detection

```
sf::Vector2f oldpos;
	int num_steps = 3;

	for (int i = 0; i < num_steps; i++) {
		oldpos = sf::Vector2f(this->_position);
		this->_position.x += this->_velocity.x * (dt/num_steps);
		_sprite.setPosition(this->_position);
		bool collision = this->_levels->at(this-
>_currentLevel).collision(_sprite.getGlobalBounds());
		if (collision) {
			this->_position = oldpos;
			_sprite.setPosition(this->_position);
		}

		oldpos = sf::Vector2f(this->_position);
		this->_position.y += this->_velocity.y * (dt/num_steps);
		_sprite.setPosition(this->_position);
		collision = this->_levels->at(this-
>_currentLevel).collision(_sprite.getGlobalBounds());
		if (collision) {
			this->_position = oldpos;
			this->_velocity.y = 0;
			_sprite.setPosition(this->_position);
		}
	}
```

Code Snippet 4 Collision detection between level and player. Calculating position vector x and y independent, to create the feel of the player movement that is desired.

10.1.10NNControlledPlayer::controllersViewOfLevel()

```
//given the position and current level the the entity is currently in return a list
of values regarding the solid state of the tiles around around the entity
std::vector<float> NNControlledPlayer::controllersViewOfLevel() const
{
	int x_tile = int(this->getSpriteCenterPosition().x / TILE_SIZE) *
int(TILE_SIZE);
	int y_tile = int(this->getSpriteCenterPosition().y / TILE_SIZE) *
int(TILE_SIZE);

	sf::FloatRect _view = sf::FloatRect(x_tile - (_left*TILE_SIZE), y_tile -
(_up*TILE_SIZE), ((_right + _left + 1)*TILE_SIZE) - TILE_SIZE / 10, ((_up + _down +
1)*TILE_SIZE) - TILE_SIZE / 10);

	std::vector<float> tileValues = std::vector<float>();
	//get to the pos of the entity in the grid position of the level
	std::vector<Tile*> tilesInArea = this->_levels->at(this-
>_currentLevel).getTilesInArea(_view);
	float value = 0.0f;
	//assign the value of the tile to a number for the controllers perception
	for (int i = 0; i < (int)tilesInArea.size(); i++) {
		if (tilesInArea.at(i)->isSolid()) {
			value = 10.0f;
		}
		else {
			switch (tilesInArea.at(i)->getTileID())
			{
			case BOTTOMOFLEVEL_TILE:
				value = -10.0f;
				break;
			case SPIKE_TILE:
				value = -10.0f;
				break;
			case CHECKPOINT_TILE:
				value = 0.0f;
				break;
			case FINISH_LINE_TILE:
				value = 0.0f;
				break;
			default:
				value = 0.0f;
				break;
			}
		}
		tileValues.push_back(value);
	}
	return tileValues;
}
```

Code Snippet 5 Function that returns an encoded array of floating point values that correspond to an array of tiles that are intersecting with the controllers view, centred around the tile that the player is in.

10.1.11Level::splitLevel()

```
//given this level return an array of the sub levels that make it up;
std::vector<Level> Level::splitLevel()
{
	std::vector<std::vector<std::vector<std::string>>> sections = this-
>chromosomeToSections();
	std::vector<Level> splitLevels = std::vector<Level>();
	std::vector<Tilemap> tilemaps = std::vector<Tilemap>();
	std::vector<int> sectionWidths = std::vector<int>();
	int w = 0;
	int addiationalColumns = 3;
	for (int i = 1; i < int(sections.size() - 1); i++) {
		//need to get the width of the the sections

		if (i == int(sections.size() - 2)) {
			addiationalColumns--;
		}

		w = int(sections.at(i).size()) + addiationalColumns;

		sectionWidths.push_back(w);
		tilemaps.push_back(Tilemap(this->_height * w));
	}

	for (int i = 0; i < this->_height; i++) {
		int sectionNum = 0;
		for (int s = 1; s < int(sections.size() - 1); s++) {
			std::vector<std::vector<std::string>>& columns = sections.at(s);
			for (int j = 0; j < int(columns.size()); j++) {
				//need the buffer vecotor
				int tileID = std::stoi(columns.at(j).at(i));
				sf::Sprite spriteTile;
				spriteTile.setTexture(this->_data-
>assetManager.getTexturesheet(TILES).getTexture(tileID));
				AssetManager::rescale(spriteTile, ZOOM_FACTOR);
				//change the width and height scaling
				sf::Vector2f pos(j*TILE_SIZE, i*TILE_SIZE);
				spriteTile.setPosition(pos);
				tilemaps.at(sectionNum).at(i*sectionWidths.at(sectionNum)
+ (j+1)) =  Tile(tileID, spriteTile, Tile::getIfSolid(tileID));
			}
			sectionNum++;
		}
	}


!!CODE CONTINUES ON THE NEXT PAGE!!
```

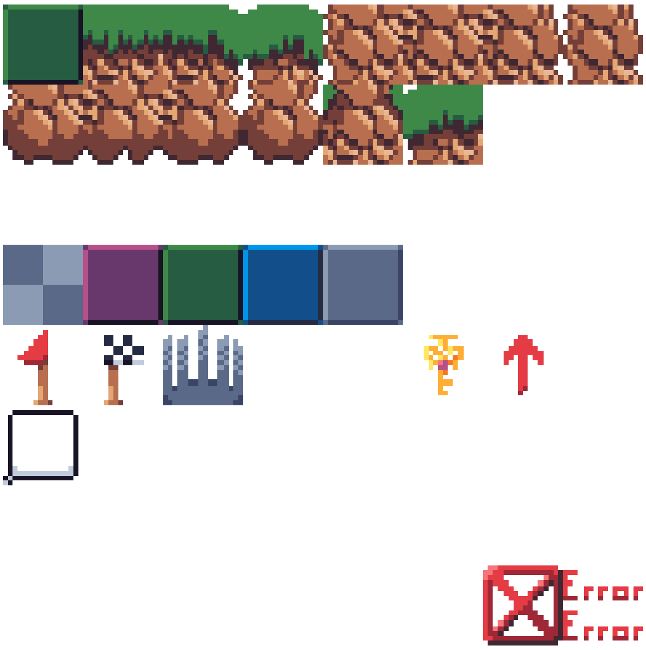
```
!!CODE START ON THE PREVIOUS PAGE!!

//copy the start of the next section to the end of the last section if it is no
the last section

	for (int i = 0; i < this->_height; i++) {
		int sectionNum = 0;
		for (int s = 1; s < int(sections.size() - 2); s++) {
			std::vector<std::string>& firstColumn =
sections.at(s+1).at(0);
			//need the buffer vecotor
			int tileID = std::stoi(firstColumn.at(i));
			if (tileID == 32) tileID = 33;
			sf::Sprite spriteTile;
			spriteTile.setTexture(this->_data-
>assetManager.getTexturesheet(TILES).getTexture(tileID));
			AssetManager::rescale(spriteTile, ZOOM_FACTOR);
			sf::Vector2f pos((sectionWidths.at(sectionNum) -
2)*TILE_SIZE, i*TILE_SIZE);
			spriteTile.setPosition(pos);

			tilemaps.at(sectionNum).at(i*sectionWidths.at(sectionNum) +
(sectionWidths.at(sectionNum)-2)) = Tile(tileID, spriteTile,
Tile::getIfSolid(tileID));
			sectionNum++;
		}
	}

	int tileID = 60;
	sf::Sprite bufferSprite;
	bufferSprite.setTexture(this->_data-
>assetManager.getTexturesheet(TILES).getTexture(tileID));
	AssetManager::rescale(bufferSprite, ZOOM_FACTOR);

	for (int i = 0; i < int(tilemaps.size()); i++) {
		for (int y = 0; y < this->_height; y++) {
			//change the width and height scaling
			sf::Vector2f pos(0*TILE_SIZE, y*TILE_SIZE);
			bufferSprite.setPosition(pos);
			tilemaps.at(i).at(y*sectionWidths.at(i) + 0) = Tile(tileID,
bufferSprite, Tile::getIfSolid(tileID));

			pos = sf::Vector2f((sectionWidths.at(i)-1)*TILE_SIZE,
y*TILE_SIZE);
			bufferSprite.setPosition(pos);
			tilemaps.at(i).at(y*sectionWidths.at(i) +
(sectionWidths.at(i) - 1)) = Tile(tileID, bufferSprite,
Tile::getIfSolid(tileID));
		}
		splitLevels.push_back(Level(_data, tilemaps.at(i),
sectionWidths.at(i), this->_height, "Resources/temp/level", 10.0f));
	}
	return splitLevels;
}
```

Code Snippet 6 Level Split function, returns an array of levels that are subsections of a given level. Used to mutate and crossover two individual levels. This function ensures the integrity of the sub sections and creates levels from the tilemaps that are gathered from the main Level.

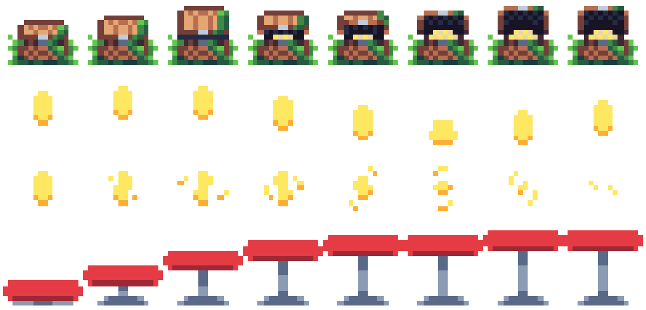
Art/Texture sheets

10.1.12Menu Background

10.1.13Main Tilesheet

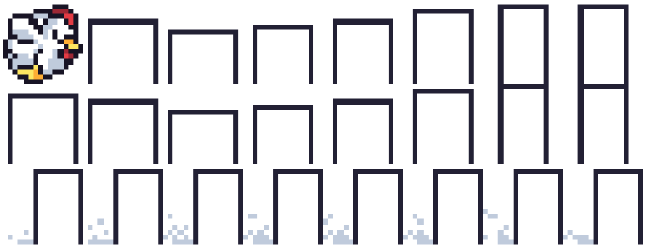


10.1.14Extra sheet



10.1.15Splashscreen

10.1.16Player Sheet



11 Project plan

UG Project Plan
CSC-30014

Project Overview and Description

Student Name: James Copping
Student Username: w4f12
Student Number: 15004812
Degree Title: BSc Computer Science
Supervisor Name: Alastair Channon
Project Title: Using a Genetic and Coevolutionary Approach to Generate 2d platformer levels built around the player. WIP (just bad)

Please provide a brief Project Description:

This project is being carried out to explore the viability of a novel content generation approach for the design and layout of levels in a 2d platforming game. The first part of the project is the game. Which will be designed and implemented from the ground up, using the c++ library SFML (Simple and Fast Multimedia Library). The process of this step will mirror the prototyping model for the base framework of the game. The design and implementation of the level generation will be the second and larger part of the project. Where levels will be created by a collection of genetic and coevolutionary algorithms influenced by how the player is plays the levels and the sections within each level.

What are the aims and objectives of the Project?

- Design and implement a fully operational game from the ground up using just a simple multimedia library.
- Design, implement, test and analyse a level generation system for the game. Using a novel approach with genetic and coevolutionary algorithms.
- Have the level generator produce a *fun and playable* sequence of levels that any player can enjoy (reliably).
- Fun and playable meaning a challenge that is possible for the player to complete while incorporating fun aspects of the base game.

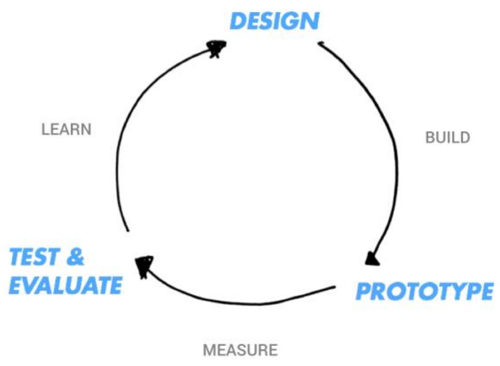
Please provide a brief overview of the key literature related to the Project:

Level Generation for MarIO Game – (Shaker, 2010) This paper goes into detail about how a level can be generated in the super Mario game, a then be evaluated by the players actions and then changed based on that. This links closely to my project and I will be taking some inspiration from this article.
Towards a Generic Framework for Automated Video Game Level Creation - (Pasquier, 2010) more really of the above paper but a more general view.
The Evolution of Fun: Automatic Level Design through Challenge Modeling (Pasquier, 2010)
Genetic Algorithms: Concepts and Applications (Man, 1996) this is being used a guide when designing the level generation system.
Real-Time Neuroevolution in the NERO Video Game (Stanley, 2005)

Project Process and Method

Please provide a brief overview of the Methodology to be used in the Project (inc. an overview of best practice within the Methodology):

Prototyping modelling



Which Data Collection Methods will be employed (e.g card sorts, questionnaires, simulations, …)?

The only data that will be collected throughout the project will be data generated by the programs that I have designed for testing and development and analysis within the project.

There may be a point in the project where I can get feedback from people who I can get to play the game and comment on anything they want to. As if it was a play test (this is only if the main part of the project is completely satisfactory).

Briefly describe how you will ensure your project is in line with BCS Project Guidelines (BSc Computer Science Single Honours Students only)?

I will be producing a piece of software that involves advanced practical programming and problem solving. Which is designed, implemented, tested and documented through a proper development life-cycle.

Time and Resource Planning

Will Standard Departmental Hardware be used? YES

If NO please outline the Hardware/Materials to be used:

N/A

Will Software which is already available in department be used? YES

If NO please outline the Software to be used including how any necessary licences will be obtained:

N/A

Will the project require any Programming? YES

If YES please list the (potential) Programming Languages to be used (including any IDEs and Libraries you may make use of):

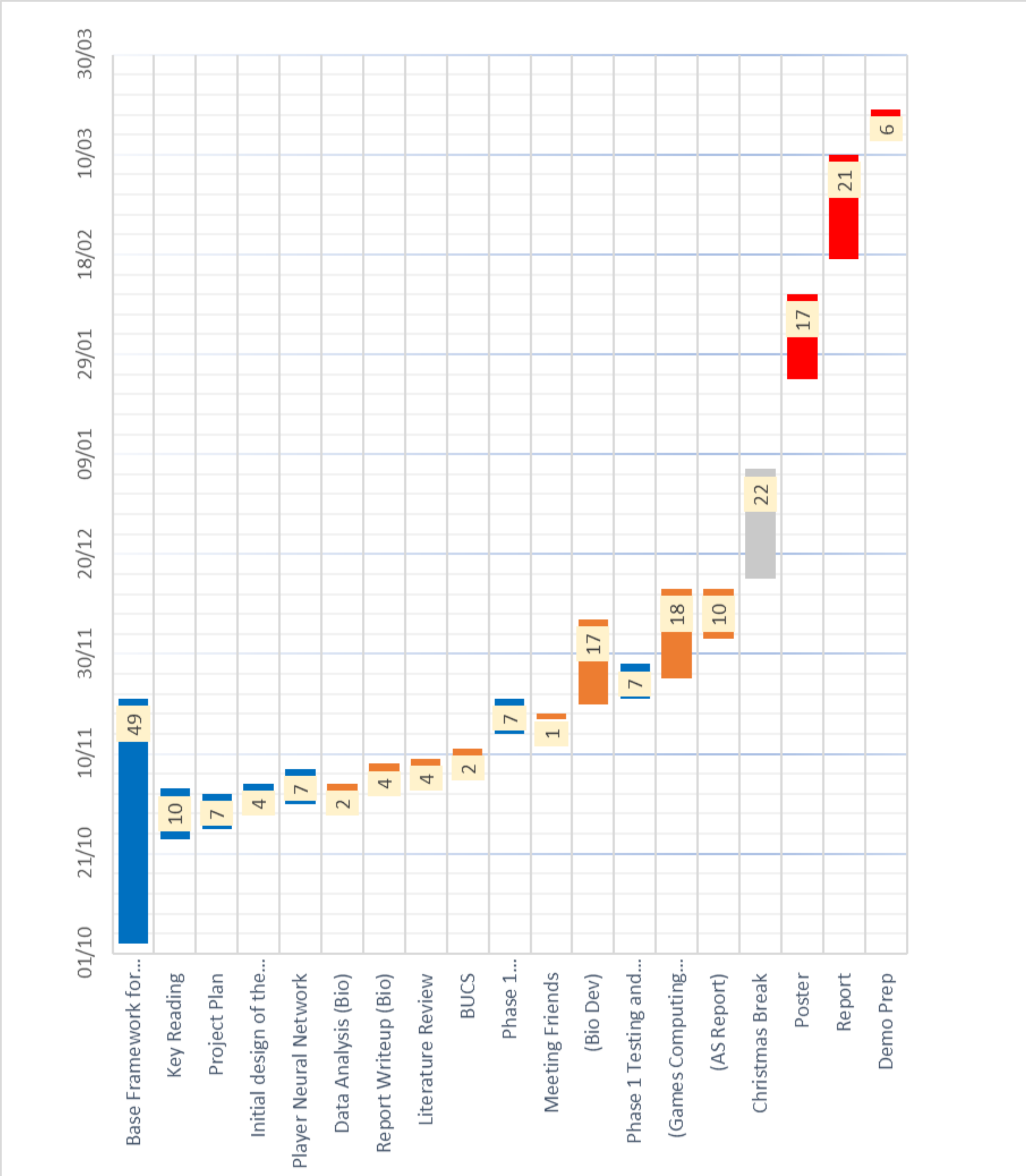
C++, Visual Studio 2017 Community, SFML (https://www.sfml-dev.org/)

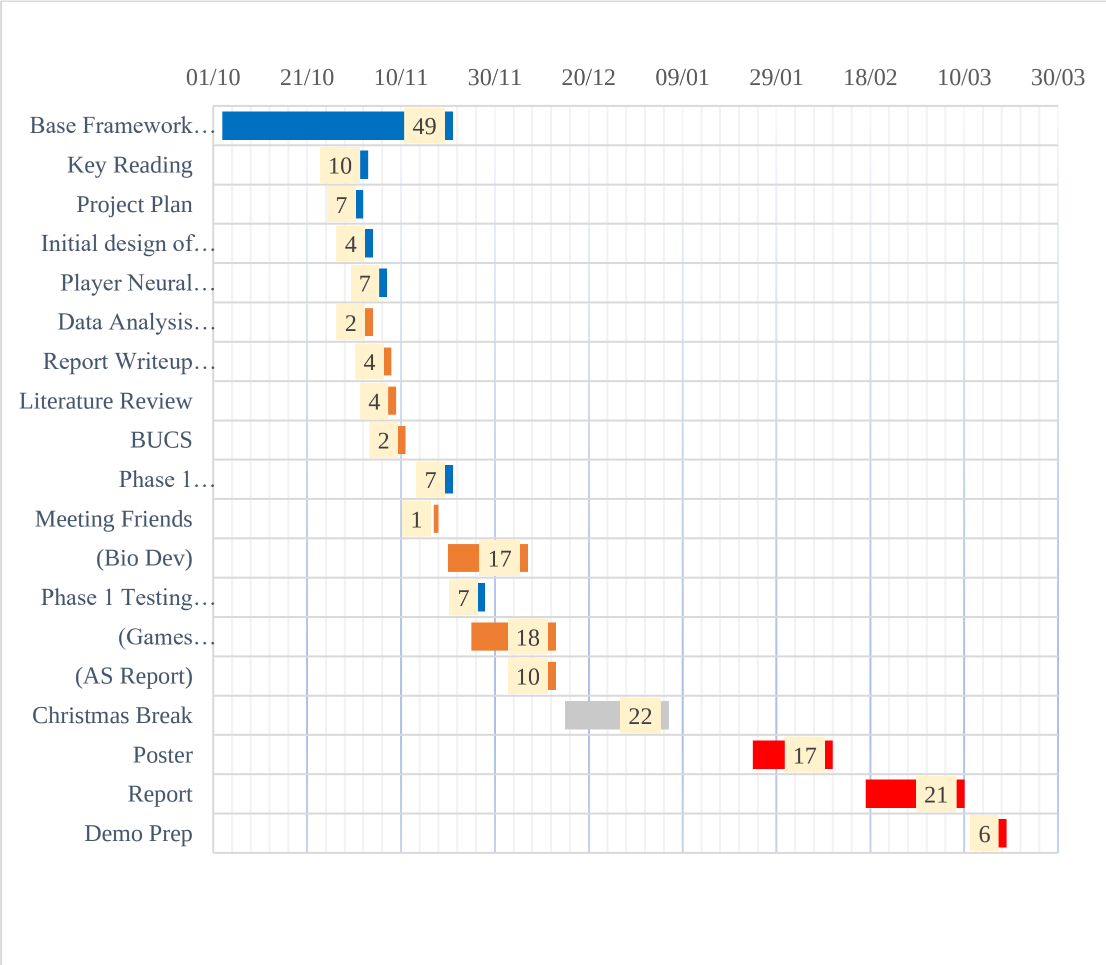
Table of Risks (*if non Standard Hardware and/or Software to be used please include backup options/ contingency plans here)*:

Risk Description	Probability	Impact	Comments
Loss of Data (neural networks and connections)	**Very Low**	**Medium - High**	**Need to make sure I have backups of the data and networks I train, or it could be time consuming to run a training phase again.**
Sfml library becomes unusable for some reason	**Very very low (sfml is a very well-kept library with plenty of documentation)**	**High**	**Use an older version or switch to SDL, this would be a huge set back.**

Gantt Chart/ Pert Chart (must include milestones and deliverables):

Project Tasks				
Start Date	**End Date**	**Description**	**Duration (days)**	**Status**
03-Oct	21-Nov	Base Framework for the game	49	working
24-Oct	03-Nov	Key Reading	10	working
26-Oct	**02-Nov**	Project Plan	7	working
31-Oct	04-Nov	Initial design of the level gen	4	working
31-Oct	07-Nov	Player Neural Network	7	working
02-Nov	04-Nov	Data Analysis (Bio)	2	stuck
04-Nov	08-Nov	Report Writeup (Bio)	4	stuck
05-Nov	09-Nov	Literature Review	4	
09-Nov	11-Nov	BUCS	2	
14-Nov	21-Nov	Phase 1 Implementation	7	
17-Nov	18-Nov	Meeting Friends	1	
20-Nov	07-Dec	(Bio Dev)	17	
21-Nov	28-Nov	Phase 1 Testing and Design Phase 2	7	
25-Nov	13-Dec	(Games Computing Coursework)	18	
03-Dec	13-Dec	(AS Report)	10	
15-Dec	06-Jan	Christmas Break	22	
24-Jan	10-Feb	Poster	17	
17-Feb	10-Mar	Report	21	
13-Mar	19-Mar	Demo Prep	6	


Base Framework for...
Key Reading
Project Plan
Initial design of the...
Player Neural Network
Data Analysis (Bio)
Report Writeup (Bio)
Literature Review
BUCS
Phase 1...
Meeting Friends
(Bio Dev)
Phase 1 Testing and...
(Games Computing...
(AS Report)
Christmas Break
Poster
Report
Demo Prep
01/10
21/10
10/11
30/11
20/12
09/01
29/01
18/02
10/03
30/03
49
10
7
4
7
2
4
4
2
7
1
17
7
18
10
22
17
21
6


01/10
21/10
10/11
30/11
20/12
09/01
29/01
18/02
10/03
30/03
Base Framework…
Key Reading
Project Plan
Initial design of…
Player Neural…
Data Analysis…
Report Writeup…
Literature Review
BUCS
Phase 1…
Meeting Friends
(Bio Dev)
Phase 1 Testing…
(Games…
(AS Report)
Christmas Break
Poster
Report
Demo Prep
49
10
7
4
7
2
4
4
2
7
1
17
7
18
10
22
17
21
6

References and Administration

Please include a list of References used in this Plan:

12 References

Man, K. F., 1996. Genetic Algorithms: Concepts and Applications. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, 43(5), pp. 1-16.

Pasquier, N. S. a. P., 2010. The Evolution of Fun: Automatic Level Design through Challenge Modeling.

Pasquier, N. S. a. P., 2010. Towards a Generic Framework for Automated Video Game Level Creation. *School of Interactive Arts and Technology, Simon Fraser University Surrey*, pp. 1-11.

Shaker, N., 2010. Level Generation Track. *The 2010 Mario AI Championship*, pp. 1-16.

Stanley, K. O., 2005. Real-Time Neuroevolution in the NERO Video Game. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 9(6), pp. 1-16.

Submission Date: 02/11/2018

13 Approved ethics forms

Student Name: James Copping
Student Username: w4f21
Student Number: 1500 4812

Proposed Project Title: Evolutionary Systems in Games Design

Project Description: The project will explore if it is feasible and appropriate to design certain aspects of a 2D game using a combination of evolutionary algorithms and neural networks. This includes AI agents within the game and how they react to the player as they learn and play the game. I also want to see how this design paradigm will affect the game in terms of: Atmosphere, Feel (Balance), Difficulty and general gameplay.
The project will consist of the full design and implementation process of a 2D 'Roguelike' Strategy Game. This will include design topics of: Narrative, Art, Sound, World/Map and Importantly AI. With a final product of the fully functional game.

Hardware and Software Requirements: Java or Cpp with opengl library

Module (delete as appropriate): CSC-30014 (30 credit)

Project Sponsor: Alastair Channon

Sponsor Signature:

Student Signature:

Date: 07/06/2018