Final Year Project Report

Learning to Play Wolfenstein3D

Gearóid Ní Ghiolla Coinnig

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Prof. Arthur Cater

UCD School of Computer Science
University College Dublin

March 26, 2017

Project Specification

General Information:

The goal is to develop a program able to play some early stages of the first person shooter video game "Wolfenstein", by learning to associate possible actions (moving in various ways, shooting) with what is perceptible in the game at any moment. A technique that proved successful in learning to play a Mario Bros level is to be applied to the new game. An existing open-source reimplementation of Wolfenstein will need to be adapted to provide the learner with information about what it perceives, where it is, how much ammunition it has, and how much time has elapsed.

The technique to be used for learning to play is a combination of neural network and genetic programming. Input neurons correspond to the presence of visible or measurable features (walls, enemies, power-ups, clock), and output neurons correspond to controller buttons (go left/right/forward/back, turn left/right/upward/downward, jump, shoot). By starting with a minimal network, adding random links from inputs to hidden nodes and onward to output nodes or perhaps other hidden nodes, and randomly adding new hidden nodes, different behaviours are obtained. By applying ideas of genetic algorithms, many individuals can be created and rated in terms of how much progress they make before dying and how soon they die. Mutation of, and crossover among, the more successful individuals of a generation leads over time to general improvement and ultimately, it is hoped, a really excellent player.

This technique succeeded in a Mario Bros game level, using inter-neuron links that had simple weights: excitatory or inhibitory. The Wolfenstein game has several similar characteristics, being deterministic and possessing something that can be used as a measure of progress (in Mario, a combination of distance from start and time taken was used but coins gathered were ignored).

Mandatory:

- Install the open-source reimplementation of Wolfenstein.
- Identify and implement code changes necessary to determine whether the hero has died, and if so, at what time and distance from starting.
- Identify and implement code changes necessary to allow a program rather than a human to control the character's actions.
- Identify and implement code changes necessary to allow a program to detect what is visible to the character at any moment in play.
- Design and implement a system for linking measurements of what can be detected in several of the floors of the first stage of Wolfenstein to activation of the player controls, using a small randomly generated system of neurons (nodes) and links that are either excitatory or inhibitory. (Such a system is virtually certain to die quickly.) Only small numbers of links in to or out from any node should be permitted at this stage, a maximum of six.
- Develop a way to combine parts of one random network with parts of another.

Discretionary:

- Design and develop a metric for comparing the degree of success of two networks, in terms of (large) distance travelled and (fast) time taken.
- Design and develop an evolutionary mechanism for taking a generation of several individual networks, picking the best few, performing crossovers and occasional mutations (new hidden nodes, wholly new links) in order to create a new generation of individuals.
- Apply this mechanism for at least 20 generations each consisting of at least 12 individuals. Measure the performances of the best, worst and median individuals in each generation.
- Apply the entire system to a third of the levels in the first Stage of Wolfenstein.

Exceptional:

- Apply this mechanism to substantially more generations, or substantially larger generations, or with more generous limits on in-degree and out-degree of nodes. Measure performance.
- Enrich the measure of performance, for example to reward the kills of enemies and the low use of ammunition.
- Apply the entire system to half or more of the levels in the first Stage of Wolfenstein.

Abstract

Wolfenstein3D is a first person shooter MS-DOS game that was released in 1992. The goal of the video game is to escape Castle Wolfenstein, a Nazi prison. Its creators, ID Software, released the source port for the game in 1995 meaning it is now possible to edit the source code for our own purposes.

The aim of the "Learning to Play Wolfenstein3D" project is to replace a human player with a computer that progressively learns to play Wolfenstein3D using two Artificial Intelligence concepts, Genetic Programming and Neural Networks. Henceforth, the AI bot created as a result of this project shall be dubbed Doop (A Developing Object-Orientated Program, also because I like the name Doop)

Doop's learning process will be aided by a weighting function that measures the success of a particular attempt that it has made at playing the game which is a requirement for any project of this nature. By using previous attempts a new, hopefully better, attempt at playing the game will be generated. The end goal being that Doop is a fully functioning AI that is capable of learning how to play some levels of the Wolfenstein3D campaign

Acknowledgments

I wish to express my sincere gratitude to the various community forum pages that have aided me in setting up such a strong foundation for my project. These include but are not limited to the DoomWorld forums, DRD team forums, StackOverflow, Wolfenstein3D Dome and the Wolf3D haven forums.

I would like to personally thank Iona Chera for providing me with information and feedback on my initial project concepts. I'd also like to thank him for outlining various ways in which I could work with the source code for Wolfenstein3D.

I want to thank my fellow classmates Joe Duffin and James Keating for always finding the time to help me with problems I have had with my project. Without their help I fear that this project may not have been as complete as it is to date.

Finally I would like to thank my supervisor Prof. Arthur Cater. Over the past year he has continually provided me with support and feedback on my project and I cannot thank him enough for that.

Table of Contents

Chapter 1: **Introduction**

Chapter 2: **Background Research**

This chapter will outline in detail the discoveries that were made whilst researching numerous relevant topics of interest to the *Learning to Play Wolfenstein3D* project. The first item that will be discussed will be the game selection process; what caused Wolfenstein to be chosen over previously considered open-source games, such as Doom and Duke Nukem. However, the main topic of discussion in this chapter will be the *NeuroEvolution of Augmenting Topologies(NEAT)* algorithm; what it is and why it will be of major importance to Doop's ability to learn.

2.1 Game Selection

In order for an AI to be able to learn to play Wolfenstein it will need to be able to make decisions based on what it can see at any point in time. This would not be possible for a game whose code was not open source since it would be impossible for the AI to read from the source code to determine what its surroundings are composed of.

2.2 Wolfenstein 3-D Game Mechanics

By the very nature of this project it is of high importance to be able to understand the source code of the game in order to determine what information can and can not be used for Doop's learning. Parts of the source where attention was mainly focused include understanding the way the map is represented, how objects, such as enemies and items that can be picked up, are tracked throughout gameplay and how the game's life-cycle works. These components will form an important base for the NEAT algorithm which is discussed after this section.

2.3 NEAT Algorithm

NEAT stands for *Neuro-Evolution of Augmenting Topologies* and is based on an paper by Kenneth Stanley and Risto Miikkulainen [1]. The paper demonstrates the ability for the NEAT algorithm to learn to solve problems in quicker succession than other types of topology evolving algorithms such as *Topology and Weight Evolving Artificial Neural Networks (TWEANN)* algorithms. This algorithm consists of two very important Machine Learning techniques, Neural Networks and Genetic Algorithms.

2.4 NEAT Neural Networks

Neural Networks(NN) are a Machine Learning technique that roughly simulate the behaviour of a brain. They generally consist of artificial Neurons that are connected using artificial Synapses. The synapses in a NN typically have a weight associated with them that describes how strong a connection any two neurons have with each other.

2.5 NEAT Genetic Algorithms

Genetic Algorithms(GA) are another type of Machine Learning technique. A typical GA will specify some population which will then be modified by the GA until a terminal condition is met. In the case of this project the terminal condition will ultimately be Doop completing a level.

A summary of this chapter will go here.

Chapter 3: **Project Approach**

The previous chapter introduced the game that was used for this project and outlined the NEAT algorithm, which is at the very core of this project. This chapter will introduce the source port used throughout the project, describe the methods employed in order to identify and understand the most important source files from the Wolfenstein source port and the importance of reading other adaptions of the NEAT algorithm.

3.1 Selecting the Source Port

Wolfenstein 3-D was originally written in a combination of the C and Assembly languages. Very early on in this project it was desired to find another adaption of the Wolfenstein source so that integrating the NEAT algorithm into the source code would not involve writing in either of these languages. This was primarily because the NEAT algorithm is much easier to understand if it is implemented in an Object-Oriented language such as C++. As such, a source port implemented in C++ was sought after.

3.2 Understanding the Source Code

Once the *WOLF4SDL* source port was selected for this project the core source files needed to be identified. This was a troublesome task. No verbose documentation for the source code exists online and even though the *WOLF4SDL* source port is essentially identical to the original source, only that it is written in C++, no in-depth documentation exists even for the original source.

Understanding the source code involved reading through it and making sense of it that way.

3.3 Reading other NEAT Implementations

This project was inspired by a video that was published on YouTube about a year and a half ago. In it, the author described how he had designed and implemented an AI capable of playing and completing the first level of Mario. The source that he wrote in order to accomplish this is open-source and served as an initial reference to understand the way in which the NEAT algorithm should be implemented.

A summary of this chapter will go here.

Chapter 4: **Design Aspects**

Chapter 3 focused on some of the tasks that were completed early on in the project time frame in order to build a strong foundation for the actual implementation of the *Learning to Play Wolfenstein3D* project. This chapter discusses the reason why NEAT is an effective algorithm for this kind of project, describes how a NEATDoop neural network works and demonstrates the interaction between the Wolfenstein gameplay and the NEAT algorithm.

4.1 Why NEAT?

As introduced in Section 2.5 the population for this project consists of a host of attempts at playing the game. Evaluating all of these attempts is very time and resource consuming. As such it is vital that each attempt at playing Wolfenstein is well formed. This very requirement immediately rules out TWEANN algorithms because in order to ensure diversity in the population it often randomises the topologies of the each network or attempt in the beginning. This introduces problems where it is required to search through the population and remove poorly performing members. This is very time consuming and as it turns out, avoidable.

4.2 Understanding the NEATDoop Neural Network

The previous section details why it is important for each attempt at playing Wolfenstein be optimal. This section focuses on understanding the implications of creating a minimally connection neural network for each attempt and how they grow over time. Here, it is important to note that the way the network is built and manipulated is directly affected by the fitness function used for evaluating the success of each attempt at playing Wolfenstein.

4.3 Wolfenstein & NEAT Interaction

Now that a comprehensive description of the NEAT algorithm has been given the next step is to understand its interaction with the Wolfenstein gameplay. This will be the focus of this section.

A summary of this chapter will go here.

Chapter 5: **Detailed Design and Implementation**

Chapter 4 gave a high level description for some of the design aspects of the *Learning to Play Wolfenstein* project. These aspects included the reasoning behind the use of the NEAT algorithm and the interaction between Wolfenstein's gameplay and the actual NEAT algorithm. The main focus of this chapter is to describe how NEATDoop interacts with components of the source code of Wolfenstein and how, as a result of this interaction, it is able to identify objects during gameplay and make decisions based on what it can see. This will also lead to a discussion on how NEATDoop's learning process is aided by assigning every attempt with a measure of success and also how an attempt at playing the game is able to be replayed.

5.1 Giving NEATDoop Game Vision

In order for NEATDoop to make decisions during gameplay it needs to be aware of its surroundings during gameplay. In order to do this it is necessary to identify how and where objects in the game are represented. As was mentioned in Section 3.2 there are *structs* in the Wolfenstein source that describe enemies and pick-ups and *arrays* that represent walls, doors and walkable space. By accessing these data structures it is possible to provide NEATDoop with so called *Game Vision*.

5.2 Playing Wolfenstein with NEATDoop

One of the main reasons for using NEAT is its efficiency. Because the every Genome starts with a simple network (only has an Input and Output layer) it is not necessary to weed out poorly performing individuals. However, this does not mean that the learning process is swift. The AI that was created by *SethBling*, in his version of this project for Mario, took 24 hours of continuous learning to complete the first level. He programmed his AI to play the game at normal speed which considerably slows down the learning process. Since Wolfenstein's gameplay is considerably more complex than Mario's it was deemed necessary to try and speed up the gameplay to hasten the learning process.

5.3 Aiding NEATDoop's Learning

In order for the AI (NEATDoop) to learn how to play Wolfenstein it is necessary for each attempt at playing the game to be assigned some sort of score that denotes the success of that attempt. This is done using a fitness function which rewards the AI for specific tasks it completes during gameplay. This section is going to outline this function and show how it can bias learning.

5.4 Attempt Termination

The terminal condition for NEATDoop is when it completes a level in Wolfenstein. This raises an interesting question though, *How do we move to the next game attempt if the AI does nothing useful?* It turns out that it is possible to use a similar approach to how *SethBling* handled this in his project. We can assign a timeout to every attempt and if the AI does nothing useful in that time frame it will timeout.

5.5 Saving NEATDoop Attempts

Chapter 6: **Testing/Evaluation**

6.1 Validating NEATDoop Basic Functionality

6.2 Analysing the Fitness Function

6.3 The Evolution of NEATDoop

6.4 Evaluating NEATDoop

Chapter 7: **Conclusions and Future Work**

7.1 NEATDoop's Ability to Learn

7.2 Extending NEATDoop

Bibliography

[1] Kenneth Stanley & Risto Miikkulainen, *'NeuroEvolution of Augmenting Topologies'*, Evolutionary Computation, vol. 10, no. 2, 2002 , 30 pp.