Neural Network Programming

With the incorporation of a genetic evolution algorithm.

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Degree Programme in Automation Engineering

Valkeakoski, Spring 2016

Martin Dominik Pinter

ABSTRACT

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Valkeakoski
Degree Programme in Automation Engineering
Option

Author Martin Dominik Pinter **Year** 2016

Subject of Bachelor's thesis Neural Network Programming

ABSTRACT

The author's aim in this project was to develop a neural network unit with the incorporation of a genetic evolution algorithm, experimenting with possibilities and the artificial recreation of neurological registration, using the programming language of C++.

The neural network evaluated input data into output data in a form to provide artificially intelligent response over different scenarios. This neural network was aided by the genetic evolution algorithm to distinguish bad results from good ones.

The author designed a simulation for the neural net to handle. This simulation tests solutions for an interception problem, running two rockets: a 'Bandit' which aims to shoot out its target and an 'Interceptor' which attempts to prevent this by eliminating the Bandit in time.

Many different scenarios were tested over different versions of the program, with the AI module reacting in different ways. The study showed that it is important to fitness calculation methods independent of expected mechanics.

The author implemented object orientated programming principles in order to make the code easily extendable for versatility and future development.

The project succeeded, the Interceptor has learnt to shoot the Bandit within different circumstances.

Keywords Neuron, neural network, artificial intelligence (AI), genetic evolution algorithm, fitness score.

Pages **36** p. + appendices 39 p.

CONTENTS

1 INTRODUCTION

It is thanks to these wondrous times, that technology is developing exponentially. Its spread is transforming workplaces and changing the methodology of productivity. As automation is rising, less tasks are getting carried out by traditional human labour, while new markets open for developers allowing to create more, to maintain, and to further possibilities.

Although technology experts have been experimenting with artificial intelligence for decades, we are yet to see the birth of a true AI. Currently revealed technologies are not more than mere imitations of how we perceive the recreation of intelligence as humans. These solutions all root to the principles of binary comparison, which is how computers are built today. The possibility to develop true artificial intelligence with today's computational architecture remains arguable.

Many technological innovations were inspired by observations made in the nature itself, of how it is constantly finding a way around its environment. Robotic vehicles built with legs resembling arthropods are being sent to extra-terrestrial planets to move on uncertain terrain. Lenses are built to adjust the focus and direction of light sources, such as the human eye would do. How wings are being used in aviary, how energy can be used as a source of light, or the way needles inject micro-organisms to blood (even the existence of these micro organizations!); it is not a surprise that artificial intelligence solutions are getting based on the very way human intelligence works. The recognition of any intelligent organism is constantly expanding while being linked together to voluntary and involuntary reflexes and reactions by its neural network. In order to understand the mechanics of this project, certain terms and ideas must be established first.

2 PRINCIPLE OF NEURAL NETWORKS

An artificial neural network is how the functions of the human brain are experimentally recreated in computing. A neuron is essentially a unit which receives signals via multiple junctions of its dendrites, called synapses. These serve as the inputs of the neuron. Every neuron has a certain threshold value, which if exceeded by the sum of the inputs the neuron shall 'fire' a signal forward, possibly to another neuron and so on, as seen on Figure 1 (Mat Buckland, AI-Junkie 2003).

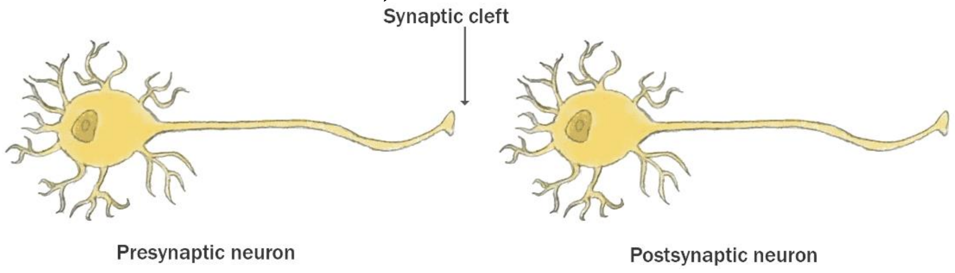


Figure 1 The feedforward connection of two neurons is illustrated, where one's output joins the other's synapse.

2.1 Modelling an artificial neuron

The structure and function of a traditional neuron may be imitated by computational connections as seen on Figure 2. As current computers calculate in binary numbers, each input and output of the machine can be essentially broken down to a series of 0s and 1s, representing either 'false' or 'true' respectively.

When modelling an artificial neuron, these may represent different input and output signals between their state being OFF or ON. With today's technology it is of no surprise, that developers are not limited to the use of values limited to binary polarity, but are able to compute with decimal numbers. However trivial it sounds, using these it is possible to represent values of different depths linking these values together allows a wide range of computing which is to be explained later.

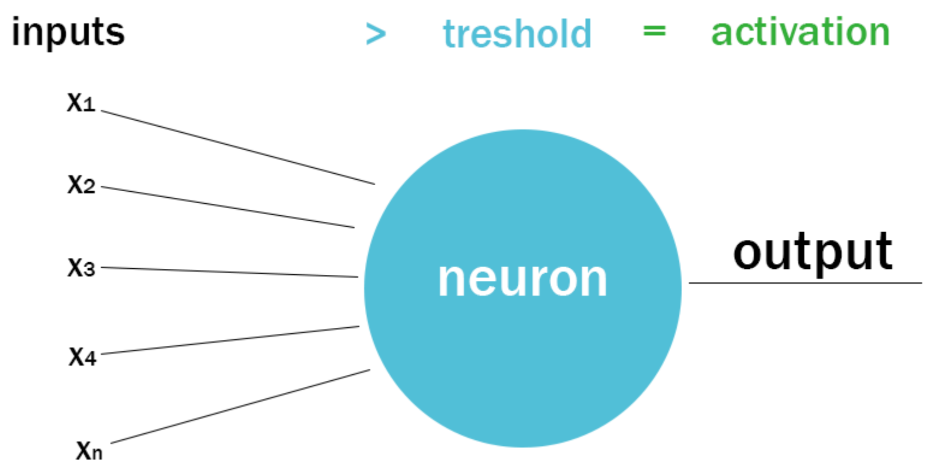


Figure 2 Basic model of an artificial neuron

2.2 Sampling the neuron's output

Although the range of inputs may vary with different values of numbers, this current model is only capable of giving a binary output step signal depending on whether the sum of its inputs did or did not exceed the threshold value of the neuron. In order to diversify the neuron's output value into a decimal range, a sigmoid function is used on the sum of inputs as seen on Figure 3.

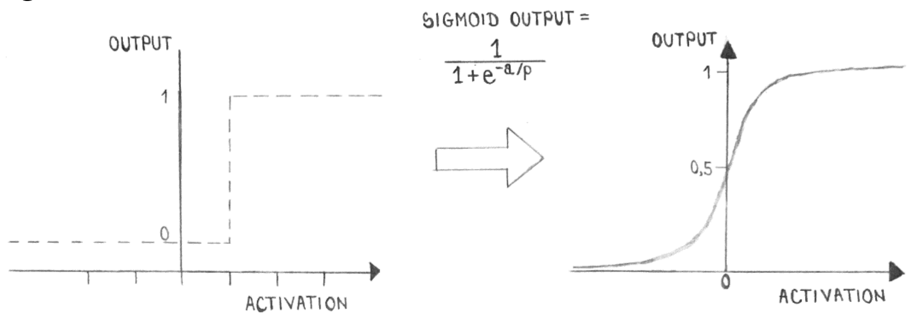


Figure 3 A step signal transfers to a sigmoid signal to give diversified values a = activation, p = response constant

This sigmoid function gives transitional values between 0 and 1, with 0.5 being on the y-axis. The diversification of the signal may be adjusted by changing the response constant 'p', transforming the signal's shape towards or against to the likes of a step function.

The author used a programming friendly version of this function, as it gives equivalent results while using less resource hungry mathematical operations for computer to handle. Using the fast sigmoid function saves time over the repeated course of the function's call, which adds up to great numbers considering that the sigmoid function is one of the highly used functions of the system.

The author's implementation of the fast sigmoid function:

```
float Sigmoid(float activation, float response) {

    activation *= response;

    float fastSigmoid = (activation / (1 + abs(activation)));

    return (fastSigmoid + 1) / 2;
}
```

The declaration of this function located in 'Utilities.h' enables default initialisation of the 'response' parameter. It allows the function to be called by only specifying the 'activation' parameter, in which case it initialises 'response' with the value of 1 to work easily with most common scenarios.

2.3 Weighted input neuron model

To allow the artificial intelligence to learn dynamic handling of various situations, weights must be added to each input of every neuron. Without weighting, the neural net would consider all the inputs at the same priority level, where as in real situations that is not the case.

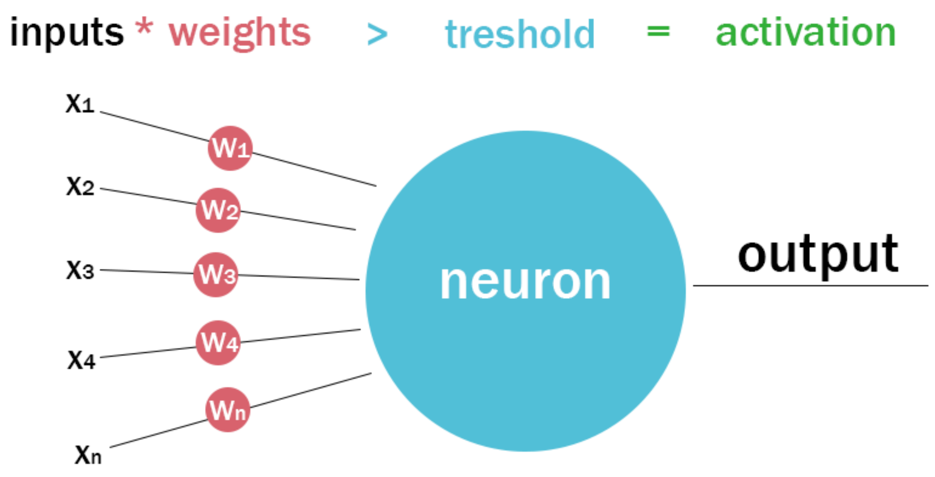


Figure 4 Model of an artificial neuron utilising weighted inputs

This way different inputs will have different impacts on the decision of the neuron, whether it should activate forward or not. Some inputs will even subtract from the common sum, provided that the input's corresponding weight is negative, as seen on Figure 4.

The value of activation may be calculated by the sum of all inputs multiplied by their corresponding weights:

$$a = \sum_{i=0}^{i=n} x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + \cdots + x_n w_n$$

For determining and handling the values of the weights, the author incorporated a genetic evolution algorithm. The definition and function of such algorithm is explained later in this work.

Artificial neurons are defined in the system by the following structure:

```
struct sNeuron {

    std::vector<float> weights;

    sNeuron(int nrOfInputs);
};
```

One neuron needs only to be aware of its weights, as this also equals to the number of its inputs, while its constructor is designed to be called by the number of expected inputs as these two values are interchangeable and parametrising the latter one gives much more sense in the later context of the code, upon calling this function.

2.3.1 Handling the threshold value

From the equation of activation concludes the equation of the neuron's decision for firing upon exceeding the threshold value 't':

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + \cdots x_n w_n \geq t$$

By subtracting the value of the threshold from both sides, it was possible to handle the threshold as a weight of a constant input of -1.

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + \cdots x_n w_n - 1 * t \geq 0$$

This constant input of -1 is often referred to as the bias in neural network studies. This way each neuron got initialised with one extra weight and the genetic evolution algorithm handled the calculation of the threshold value automatically.

2.4 Structure of a neural net

A neural net consists of different layers as seen on Figure 5:

- The input layer is located at the bottom of the chain, the inputs of these neurons serve as the senses of the neural network. The value of every transferred output will essentially be based on these in the future.
- Hidden layer(s) serve to offer more variations on the outcome of the neural net. The numbers of these layers can vary from 0 to any number desired, although it is common to have a value of at least 1.
- The output layer is the what finally becomes of these series of inputs. The values of these are ultimately translated as actions or decisions of the artificial intelligence.

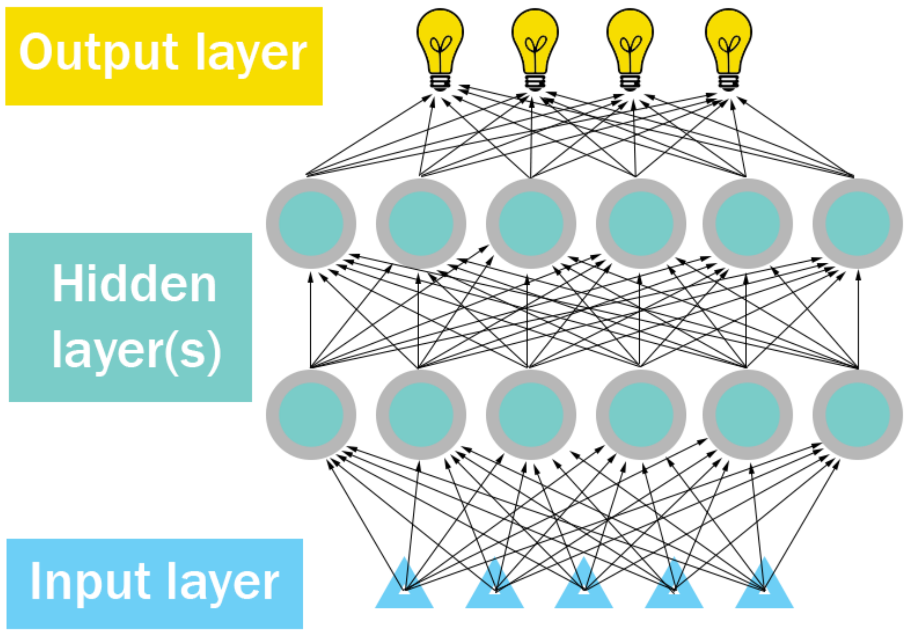


Figure 5 Modelled structure of an artificial neural network

2.4.1 Defining neuron layers to the program

The code defining neuron layers covers only the hidden layers and the output layer, as the input layer remains invisible, since the input values are simply passed from a separate function which gathers them.

```
struct sNeuronLayer {

    int nr0fNeurons;
    std::vector<sNeuron> one_neuronLayer;

    sNeuronLayer(int nr0fNeurons, int nr0fInputsPerNeuron);
};
```

This structure of a neuron layer has member variables containing the number of its neurons, and a vector full of its neurons, representing one neuron layer. Its constructor must be called with parametrising the number of neurons to be allocated, and the number of inputs per each neuron.

2.4.2 Neural net as an object

An instance of a neural net is defined in the following class:

```
class NeuralNet {
private:

    std::vector<sNeuronLayer> all_neuronLayers;

public:

    NeuralNet() { createNet(); }

    void createNet();

    void feedWeights(std::vector<float> inputWeights);

    std::vector<float> evaluate(std::vector<float>);
};
```

The contents of the neural net, in other words its neuron layers are stored as private members of the class, indicating that they should not be directly modified by architectural reasons.

The default constructor calls the 'createNet()' function which builds the structure of the neural net according to parameters specified in the 'Params' class, such as the number of inputs, number of hidden layers, the number of neurons per hidden layer, et cetera.

```
void NeuralNet::createNet() {

    if (Params::nr0fHiddenLayers > 0) {

        for (int i = 0; i < Params::nr0fHiddenLayers; ++i) {

            if (i == 0) {
                all_neuronLayers.push_back(sNeuronLayer
                (Params::NeuronsPerHiddenLayer,
                Params::nr0fInputs));
            }
            else {
                all_neuronLayers.push_back(sNeuronLayer
                (Params::NeuronsPerHiddenLayer,
                Params::NeuronsPerHiddenLayer));
            }
        }

    all_neuronLayers.push_back(sNeuronLayer
    (Params::nr0fOutputs, Params::NeuronsPerHiddenLayer));
    }
    else {
        all_neuronLayers.push_back(sNeuronLayer(
        Params::nr0fOutputs, Params::nr0fInputs));
    }
}
```

3 GENETIC EVOLUTION ALGORITHM

This algorithm simulates the evolution of numbers representing digital data, and is crucial for the neural network to evaluate present circumstances based on results, and to understand achieving better results.

The algorithm is based on how scientists perceive nature's evolution and natural selection based on the fitness of one specimen. According to evolution, countless different individuals rose from a certain species throughout billions of years and generations, from bacteria to intelligent organisms. Looking back at the roots of this process spanning through an inconceivable amount of time, from the point life on earth started out from micro-organisms to these very times where scientists are attempting to artificially recreate functional intelligence; all this change, labelled as evolution can essentially be broken down to a series of mutations and genetic deviation.

Needless to say, only a minimal amount of these deviations conclude to the top of the food chain or even prove themselves useful at all, but that one specimen which has managed to mutate into having a new sense or superior ability, could evade danger and become more attractive for reproduction, and in case it succeeds in this latter, its mutated DNA earnt the chance to be spread into the gene pool of its species, over future *generations*.

Since not all changes are beneficial and the author was attempting to virtualise such a tendency, it was important that each individual of the current generation got assessed by a *fitness score* representing how well it performed with the current set of rules, or in other words: how much did it *fit* solving the problem at hand (Mat Buckland, AI-Junkie 2003).

3.1 Methodology and terminology

The genetic evolution algorithm was designed for intuitive problem solving by numbers. First it assembles a pool of experimental solutions by generating a pool of random numbers. Each of the possible solutions – good or bad – are referred to as *genomes*, which consist of a series of *chromosomes* each, and the whole bunch of them constitute what is ultimately called a *population*.

So the algorithm tries to solve the problem at hand by substituting each *genome* in the solution, and then assigns a *fitness score* to each of them, indicating how well that certain *genome* performed against a certain set of rules, where a higher score indicates better performance. The same process is repeated for the rest of the *population* against the same set of rules, until each of its genomes gets tested and gets a fitness score assigned to their performance correspondingly.

Once each and every genome of the current population has gone through the testing phase, the algorithm reaches its evolutionary phase and executes it on its population of genomes. A new population based on the currently existing genomes is to be created, thus stepping into the next – in this case second – generation of the cycle.

To put things into perspective in correspondence to the author's project where a genetic algorithm was combined with a neural net, one chromosome represents a single weight assigned to one input of a neuron, concluding that one genome contains all the chromosomes (weights) required for one whole neural net to run, while one population contains the pool of genomes (set of weights) which is to be evolved into the next generation. Simply put, upon examining a modelled picture of the neural net (see figure 5), each line represents an input, thus a weight, thus one chromosome, where all the lines together build up one genome.

In order for the current generation's population to procreate into the next generation, the genetic evolution algorithm executes the steps described in the following sub-chapters:

3.1.1 Awarding the best genomes

Evolution starts by duplicating the best genomes into the population of a new generation. Using this function moderately ensures faster learning, while abusing it certainly hurts the outcome of the evolution.

```
std::vector<sGenome>
Population::pickBests(int topN, int copies) {

    std::vector<sGenome> returnvector;

    for (unsigned i = 0; i < topN; ++i) {

        for (unsigned j = 0; j < copies; ++j) {
            returnvector.push_back(Genomes[i]);
        }
    }
    return returnvector;
}
```

This function takes two parameters. 'topN' represents the number of places awarded, while 'copies' indicates how many copies of the genomes will persist through evolution, e.g. if 'topN' = 3 and 'copies' = 2, two copies of the three best genomes would survive.

3.1.2 Roulette wheel selection

Two parents must be selected from the current population, each of them represented by individual genomes. Note that this function assumes that the genomes are sorted beforehand, by their fitness values increasingly. The definition of the function 'SortPopulation()' is found in Appendix 24, under the file 'Population.cpp'. The author defined the selection of one parent according to the following code snippet:

```
sGenome Population::Roulette() {

    float Sum = 0.0f;
    float FitnessSoFar = 0.0f;
    for (int i = 0; i < Params::PopulationSize; ++i) {
        Sum += Genomes[i].fitness;
    }

    double Slice = (double)(randfloat() * Sum);

    for (int i = 0; i < Params::PopulationSize; ++i) {

        FitnessSoFar += Genomes[i].fitness;

        if (FitnessSoFar >= Slice)
            return Genomes[i];
    }
    return Genomes.back();
}
```

Roulette wheel selection is a method of selection, where each candidate may qualify, but those with higher fitness scores stand a better chance proportionally.

First one needs to go through all the genomes in the current population, to sum up their fitness values into a variable. Then a random decimal number generated, in a dynamically defined range. The range is dependent on the result of a previously defined function called 'randfloat()', which returns a random decimal number between 0 and 1, then this number is multiplied by the calculated sum of the fitness scores of the whole population. This will define a random point in the totaled fitness range, and the genome which occupies that piece after accumulation will get selected. Note, that this function is built to return only one genome at a time. In order to select two parents, it must run twice.

3.1.3 Genetic crossover

Genetic crossover function simulates the reproduction of two genes. The genes will interchange part of their genetic content with each other, to breed offspring resembling a variant of their mixture. The author defined it as follows:

```
std::vector<sGenome>
Population::Crossover2(std::vector<sGenome> Parents) {

    float CrossoverRate = 0.7f;

    if (randfloat() <= CrossoverRate) {

        int RandomCrossoverPoint = rand() %
        Params::WeightCount;

    for (int i = RandomCrossoverPoint; i <
    Params::WeightCount; ++i) {

        std::swap(Parents[0].value[i], Parents[1].value[i]);
    }

        std::vector<sGenome> NewParents;
        NewParents.push_back(Parents[0]);
```

```
            NewParents.push_back(Parents[1]);

            return NewParents;
        }
        else
            return Parents;
}
```

Number two in the name of the function suggests that it expects two parents. Not every parent qualifies to a crossover, the rate of this in percentages is defined in the floating point variable 'CrossoverRate' as 70%. The value of this is up to personal definition, aiming to experiment with getting better results. This function will mark a random common point on the length of the two genomes, from which onwards it will swap their genetic contents with each other. In case the parents did not qualify to crossover, the function returns them unmodified right after the crossover decision has been made without a further run of the function.

3.1.4 Mutation

Mutation may occur to chromosomes under a very small percentage. The function starts by iterating through each chromosome of the selected genomes. Once mutation takes place, a function reverses the value of that one chromosome. Since chromosomes in the author's project were decimal numbers between 0 and 1, reversing them was done by subtracting their values from 1.

```
std::vector<sGenome>
Population::Mutate2(std::vector<sGenome> Specimens) {

    float MutationRate = 0.005;

    bool ChangesDone = false;

    for (int i = 0; i < Specimens[0].value.size(); ++i) {

        if (randfloat() <= MutationRate) {

            Specimens[0].value[i] = 1 -
            Specimens[0].value[i];
            ChangesDone = true;
        }
    }

    for (int i = 0; i < Specimens[1].value.size(); ++i) {

        if (randfloat() <= MutationRate) {

            Specimens[1].value[i] = 1 -
            Specimens[1].value[i];
            ChangesDone = true;
        }
    }

    if (ChangesDone) {

        std::vector<sGenome> returnvector;
```

```
            returnvector.push_back(Specimens[0]);
            returnvector.push_back(Specimens[1]);

            return returnvector;
        }
        return Specimens;
    }
```

Note that since mutation only occurs rarely, the author designed this function so that it allocated the vector containing the mutated specimens only if they got qualified for mutation. This is for the sake of efficient memory allocation, and a dedicated Boolean variable 'ChangesDone' was declared to handle this.

4 DEVELOPMENT IN C++

4.1 History and overview of the language

The programming language C++ dates its roots back to 1979 when it was created by Bjarne Stroustrup. It was originally named 'C with Classes' as it offered object oriented programming capabilities. In 1983 it got renamed to its current name, where the ++ operator resembles the one utilised in the language for incrementing a variable's value. This hints at the constant growth of the language, which was essentially based on C. The shape of C++ has gone a long way since its baby steps. New libraries are getting added to extend the possibilities and to offer simpler built functions for more complex problems (cplusplus.com, 2016).

4.2 Object oriented development (OOD)

Object oriented programming is an architectural principle, which aims for delegated correlating developer-specified objects to correlate with each other. This way real life or abstract objects may be custom-defined by their common properties, functions and types of these. Once an object is defined, multiple instances of the same type may be allocated to the virtual memory of the computer with different values for their properties.

The basic building blocks of OOD lay in classes and structures. These two are highly similar to each other in terms of defining custom objects, with the only major difference that by default, all the member functions and variables of a structure are public, while for classes they are private. For the rest of this thesis work, the author will only refer to classes, as these are more frequent interpretations than structures and are contextually equivalent. A class may have its permission attributes defined by the three values of 'public', 'private' and 'protected' as follows:

– Public members of a class or structure may be referenced from outside the class definition, calling it attached to one instance of an object or referring by class.

- Private members may only be referred by other members of the class or structure.
- Protected members may be referred by other members of the class or structure, and by friend and inherited classes or structures of the same type.

Conventionally, classes and their member functions and variables are *declared* in their designated header files e.g. 'Object.h', while their member functions are *defined* in their cpp files e.g. 'Object.cpp'.

4.2.1 Derived classes and inheritance

A class may be derived from another class, this latter becoming the base class of the derived class. This way a derived class inherits individual instances of all the member variables and functions of the base class, as now both of the classes mutually define properties of that object.

4.2.2 Virtual functions and abstract classes

Any function may be made virtual by using the 'virtual' prefix in its declaration. This allows any derived class of the current class to inherently override the definition of the function.

A virtual function can be made into a pure virtual function by adding ' = 0' as a suffix into its declaration. A pure virtual function must only have definitions in its class' derived classes, while a virtual function is allowed to have a default definition in the same class it is declared in.

A class containing the declaration of a pure virtual function is called an abstract class. Any derived class of that class which does not individually define that/those pure virtual function(s) also becomes abstract.

A popular example for the usage and benefits of virtualisation:

```
class Animal {
public:

    int legs;
    int lives;

    virtual void eat() = 0;
};

class Dog : public Animal {
public:

    Dog() { legs = 4; lives = 1; }
    virtual void eat() override { /* Eat dog food */ }

};

class Cat : public Animal {
public:
    Cat() { legs = 4; lives = 9; }
    virtual void eat() override { /* Eat cat food */ }
};
```

As the code snippet shows, both 'Dog' and 'Cat' are derived classes of the abstract class 'Animal' so they both maintain separate instances of their member variables 'legs' and 'lives' as they get assigned to different values in their derived classes' *constructor*.

The beauty of virtual functions show when later both of them call their separately overridden 'eat()' function.

```
Dog Charlie;
Cat Oliver;

Charlie.eat();
Oliver.eat();
```

As this results in the two different objects executing different things when calling the same commonly inherited pure virtual function.

4.3 SFML library

Since the default output of the language would run in a console window, the author used the open-source SFML library to provide visual representation of the project. SFML contains new type of objects, such as graphically rendered windows, a graphical representation of objects e.g. text, shapes and sprites, new container objects and much more.

4.3.1 Disclaimer

The author takes no credit for any work done by the developers of SFML. The SFML library alone does not compile to results, only extends the range of objects available for development. The structural architecture and the contents of this Bachelor's Thesis are the genuine work of the author and do not present code written by such a third party. This concludes that no content written by the SFML developers was published here or claimed as my-own in any of this work, including the Appendices.

The licence of the SFML library is available on their website http://www.sfml-dev.org/license.php

5 ROCKET INTERCEPTION SIMULATION CLASS HIRERACHY OF OBJECTS

Each visually represented object taking part in the flow of the simulation has been defined by a chain of several classes for the sake of object orientated development. This chapter aims to describe them thoroughly since their synopsis is crucial to understanding the frame of the methodology.

Classes of visually represented objects and their hierarchy were defined by the author as seen on Figure 6.

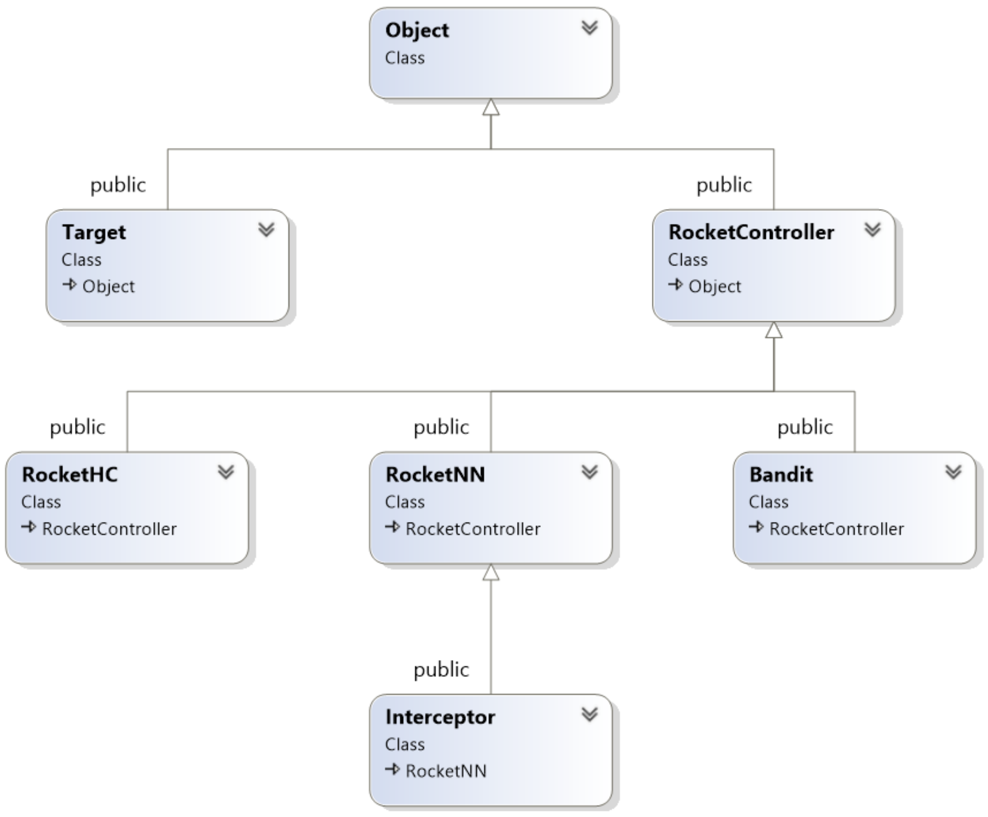


Figure 6 Diagram indicating the hierarchy of object classes

5.1 'Object' class

The aim of this class was to define common properties of different objects. It contained such member functions and variables which were used by all classes derived. It was declared as follows:

```
class Object {
public:

    Object() {}
    Object(sf::Vector2f  position,  sf::Vector2f  velocity,
float   angle,   sf::Vector2f   size,   float   scale)   :
position(position),    velocity(velocity),    angle(angle),
size(size), scale(scale) { }

    virtual void draw(sf::RenderWindow& window) = 0;
    virtual void update() = 0;

    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f size;
    float angle = 0;
    float scale;
    bool Collided = false;
```

```
        sf::FloatRect getBoundingBox();
        bool isOOB();
};
```

5.1.1 Locally declared members

An instance of this object may be constructed from any of its derived classes, which will essentially collect and pass down required data by the chain of their constructors.

<u>Constructing parameters of the class:</u>

- Position of the object defined in a vector of 2 floating point numbers as its x and y coordinates respectively.
- Velocity vector of the object defined in a vector of 2 floating point numbers as its x and y coordinates indicating the velocity vector's end points relative to its own position being the origin of its own coordinate plane.
- Angle representing the rotated angle of the object in respect to the plane of the graphical window, in radians.
- Size and Scale of the objects, which are defined for their graphical representation and collision detection functions.

<u>Virtual functions of the class:</u>

- Function 'draw(sf::RenderWindow& window) = 0' is a pure virtual function of the class to be later defined by the individual methods of different derived classes. It takes the reference ('&') of a 'sf::RenderWindow' object to draw its content to. It is important that only the reference of the window is passed, as passed parameters duplicate into new copies existing only inside the scope of the function. This would mean a second window being rendered otherwise.
- The 'update() = 0' pure virtual function will update the values of this class' member variables after one tick/frame of the simulation has passed.

<u>Local members of the class:</u>

- The Boolean variable 'Collided' stores the current state of the object, whether it has collided or not.
- The 'getBoundingBox()' function returns coordinates of the smallest possible rectangle covering all the object's visual pixels in order to calculate collision later.
- The 'isOOB()' functions returns a Boolean variable if the object's position is outside the plane of the window.

5.1.2 Notable local function

The 'isOOB()' function returns its Boolean variable based on the following one line, where the '||' operator represents the logical 'OR' gate and therefor returns the logical true/false value evaluated for the whole expression enclosed in brackets.

```
bool Object::isOOB() {

return (position.x > Params::WindowWidth
        || position.y > Params::WindowHeight
        || position.x < 0
        || position.y < 0);
}
```

The 'getBoundingBox()' function returns the coordinates of the rectangle using the 'size' and 'scale' given as parameters of the class' constructor to retrieve accurate size of the object in respect to the scaling modifier.

```
sf::FloatRect Object::getBoundingBox() {
           return (sf::FloatRect(position, size * scale));
}
```

5.2 'Target' class

This class was directly derived from the Object class as it stands to standardise target objects for the Bandit, and was declared as follows:

```
class Target : public Object {
public:
    Target() : Target(Params::posTarget, Params::radTarget,
sf::Vector2f(150.0f, 150.0f), Params::scaleTarget) {}

    Target(sf::Vector2f position, float radius, sf::Vector2f
size, float scale) : Object(position, Params::nullvec,
Params::angleTarget, size, scale), radius(radius),
TextureString(Params::TargetT) {

    BodyTexture.loadFromFile(TextureString);
    BodyTexture.setSmooth(true);
    BodyTexture.setRepeated(false);
    }

    virtual void draw(sf::RenderWindow&) override;
    virtual void update() override;

    std::string TextureString;
    sf::Texture BodyTexture;
    sf::Sprite BodySprite;
    sf::CircleShape Body;
    float radius;

    void reset();

    bool randomized = true;
};
```

5.2.1 Locally declared members

<u>Constructor parameters:</u>

The constructor of this class hands parameters to the Object class' which are common for one instance of both. The sovereign parameters of this function's constructor are:

- The size of the object defined as 'radius'.
- The location of its texture file as 'TextureString'.

  This object may be constructed from its default constructor, as then it calls the parametrized constructor based on default values defined in 'Params'.

Virtual functions:
- draw(sf::RenderWindow&)
- update()

are getting overridden after their pure virtual declaration in the Object abstract base class.

Local members:
- 'TextureString' stores the location of the image file used as the texture of this objects.
- For an object to hold the image as its texture, 'BodyTexture' is defined.
- 'BodySprite' is defined to make a sprite out of the variable 'BodyTexture'
- The 'reset()' function resets the target back to its initial properties before every new simulation.

5.2.2 Notable local function

The 'update()' function updates the position of the Target for every tick/frame of the simulation. It also redefines the centre of the object relative to the 0 point of its coordinate plane which is its sprite's top left corner.

```
void Target::update() {
    BodySprite.setPosition(position);

    BodySprite.setOrigin(Params::radTarget / 2,
    Params::radTarget / 2);
}
```

5.3 **'RocketController' class**

This class describes the physics and the movement mechanics of rockets.

```
class RocketController : public Object {
public:
    RocketController() {}
    RocketController(sf::Vector2f position, float angle,
std::string ntTexture, std::string ftTexture) :
Object(position, Params::nullvec, angle, sf::Vector2f(45,
45), Params::scaleRocket), noThrottleString(ntTexture),
fullThrottleString(ftTexture) {

    noThrottle.loadFromFile(noThrottleString);
    noThrottle.setSmooth(true);
    noThrottle.setRepeated(false);
```

```
        fullThrottle.loadFromFile(fullThrottleString);
        fullThrottle.setSmooth(true);
        fullThrottle.setRepeated(false);
        }

        virtual void controls() = 0;

        virtual void update() override;
        virtual void draw(sf::RenderWindow& window) override;

        float throttle = 0.0f;
        void accelerate(float amount);

        void DefineTarget(Object * EnemyRocket);
        Object * LockOnTarget;

        float angular_throttle = 0.0f;
        void angular_accelerate(float amount);
        float angular_velocity = 0.0f;

        float prevAngle = angle;
        float rotationalSum = 0.0f;

        sf::Color color;
        sf::Texture noThrottle;
        std::string noThrottleString;
        sf::Texture fullThrottle;
        std::string fullThrottleString;
};
```

5.3.1 Locally declared members

<u>Sovereign constructor parameters:</u>

- 'ntTexture' defines the texture of the object, in case it has no throttle.
- 'ftTexture' defines the texture of the object, in case it has full throttle.

<u>Virtual functions:</u>

- 'controls()' is declared here as a pure virtual function, which makes the RocketController class abstract. This function is defined later only by the derived classes.
- The virtual function 'update()' for overriding definition.
- The virtual function 'draw(sf::RenderWindow&) for overriding definition.

<u>Local members:</u>

- 'TargetHit()' examines if the rocket has collided, setting its 'Object::Collided' variable accordingly.
- 'throttle' represents the rocket's current throttle by a floating point decimal number between 0 and 1.
- 'accelerate(float)' accumulates the value of 'throttle', clamping the result between 0 and 1.
- 'DefineTarget(Object *)' receives an Object's pointer as a parameter, and feeds its value to the 'LockOnTarget' member variable.
- 'LockOnTarget' stores the rocket's target as a pointer to the Object class.

- 'angular_throttle' represents the throttle of the rocket in angular directions.
- 'angular_accelerate()' accumulates the value of 'angular_throttle', clamping the result between 0 and 1.
- 'angular_velocity' stores the rocket's momentary angular velocity.
- 'CheckForSpin()' tests the rocket for undesired spinning, setting the 'SpinAlert' variable accordingly.
- 'SpinAlert' is true if the rocket is spinning.
- 'prevAngle' stores the latest angle of the rocket from the previous tick/frame of the simulation.
- 'rotationalSum' is used for summing rotational movements for the 'CheckForSpin()' function.
- 'noThrottle' and 'noThrottleString' together define the texture of the rocket in case it has no throttle.
- 'fullThrottle' and 'fullThrottleString' together define the texture of the rocket in case it has throttle.

5.3.2 Notable local function

The 'draw()' function sets the rocket's parameters according to correct visual representation. It also switches between textures depending on the value of 'throttle' and calls an instance of 'RocketHUD' exclusive to the object.

```
void RocketController::draw(sf::RenderWindow& window) {

    sf::Texture* rocketTexture;

    if (throttle == 0.0f) {
        rocketTexture = &noThrottle;
    }
    else {
        rocketTexture = &fullThrottle;
    }

    sf::Sprite rocketSprite;
    rocketSprite.setTexture(*rocketTexture);
    rocketSprite.setPosition(position);
    rocketSprite.setRotation((angle + Params::pi / 2) *
     (180.0f / Params::pi));
    rocketSprite.setOrigin(19, 27);

    rocketSprite.setScale(Params::scaleRocket,
    Params::scaleRocket);

    window.draw(rocketSprite);
    RocketHUD HUD(window, *this);
}
```

5.4 **'RocketHC' class**

This class constructed a human controlled rocket derived from 'RocketController', currently used in Sandbox mode.

It offered a simple extension derived from the class of RocketController, with 'controls()' overridden.

```
class RocketHC : public RocketController {
public:
    RocketHC() {}
    RocketHC(sf::Vector2f position, float angle) :
     RocketController (position, angle, Params::BlueNTT,
     Params::BlueFTT) {}

    virtual void controls() override;
};
```

5.4.1 Locally declared members

<u>Sovereign constructor parameters:</u>

- Position of the 'RocketHC' defined in a vector of 2 floating point numbers as its x and y coordinates respectively.
- Angle in radians representing the rotated angle of the object in respect to the plane of the graphical window.

<u>Virtual functions:</u>

- Controls are getting virtually redefined.

5.4.2 Notable local function

The 'RocketHC::controls()' function calls 'RocketController::accelerate(float)' and 'RocketController::angular_accelerate(float)' functions to move and turn the rocket upon the press of the arrow keys.

```
void RocketHC::controls() {

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
        accelerate(0.01f);
    else accelerate(-0.006f);

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
        if (angular_throttle > 0)
            angular_accelerate(-0.03f);
        else
            angular_accelerate(-0.01f);
    }
    else if (sf::Keyboard::isKeyPressed(
             sf::Keyboard::Right)) {

        if (angular_throttle < 0)
            angular_accelerate(0.03f);
        else
            angular_accelerate(0.01f);
    }
    else
    angular_accelerate((-1 * angular_throttle) / 30);


if (sf::Keyboard::isKeyPressed(
 sf::Keyboard::Down))
```

```
        accelerate(-0.33f);
    }
```

5.5 **'Bandit' class**

This class constructed an adversary rocket aiming at its defined target.

```
class Bandit : public RocketController {
public:

    Bandit() {}
    Bandit(sf::Vector2f position, float angle) :
     RocketController(position, angle, Params::RedNTT,
     Params::RedFTT) {}

    void reset(int seed);
    virtual void controls() override;

    virtual void update() override;
};
```

5.5.1 Locally declared members

<u>Sovereign constructor parameters:</u>

– Position of the 'RocketHC' defined in a vector of 2 floating point numbers as its x and y coordinates respectively.
– Angle in radians representing the rotated angle of the object in respect to the plane of the graphical window.

<u>Virtual functions:</u>

– Controls got virtually redefined.
– Update got virtually redefined.

<u>Local members:</u>

– Reset got declared, initializing random generating seed parameter.

5.5.2 Notable local function

The local reset function uses Mersenne Twister pseudorandom number generator.

```
void Bandit::reset(int seed) {

    std::mt19937 gen(seed);
    std::uniform_int_distribution<> disX(
     Params::WindowWidth * 0.9, Params::WindowWidth * 0.9 +
     Params::WindowWidth / 10);

    std::uniform_int_distribution<> disY(
     Params::WindowHeight * 0.1, Params::WindowHeight *
     0.1 + Params::WindowHeight * 0.8);

    int randX = disX(gen);
    int randY = disY(gen);

    position = sf::Vector2f(randX, randY);
    velocity = Params::nullvec;

    float Egocent_x = LockOnTarget->position.x - position.x;
    float Egocent_y = LockOnTarget->position.y - position.y;
    angle = atan2(Egocent_y, Egocent_x);

    throttle = 0.0f;
    angular_throttle = 0.0f;
    angular_velocity = 0.0f;
    Collided = false;
}
```

This pseudorandom number generator method ensured unified distribution of random numbers, unlike common random functions.

5.6 'RocketNN' class

This class constructed a rocket controlled by a neural net.

```
class RocketNN : public RocketController {
public:
    RocketNN() {}

    RocketNN(sf::Vector2f position, float angle) :
     RocketController(position, angle, Params::BlueNTT,
     Params::BlueFTT) {}

    virtual void controls() override;
    virtual void update() override;

    virtual std::vector<float> getNNinputs() = 0;

    void calcDistance();
    float ClosestDistanceToTarget =
     sqrt(pow(Params::WindowHeight, 2) +
     pow(Params::WindowWidth, 2));

    float calcLookAtScore();
    float LookAtScore = 0;
    float normalizedLookAt();

    std::vector<float> NNInputs;
    float calcFitness(float SimulationTime);

    std::vector<float> NNControls;

    void SetNNControls(NeuralNet *NN);

};
```

5.6.1 Locally declared members

<u>Virtual functions:</u>

– Controls and updates are virtually overridden.

– 'getNNinputs()' is declared as pure virtual function. Derived classes will define it with their calculations concerning the inputs of their neural net. The result of this function is essentially what a rocket sees.

Local members:

– Members are calculating the history of closest distance to the target.
– Methods for calculating the rocket's 'LookAtScore' are defined.
– Control values are retrieved from the neural net by SetNNControls()
– Control values retrieved from the neural net are stored in 'NNControls'
– Fitness of the current genome controlling that instance of a 'RocketNN' is determined by the calculations of 'calcFitness(float)'

5.6.2 Notable local function

The 'normalizedLookAt()' function determined the direction of the target compared to the look-at angle of the 'RocketNN'.

```
float RocketNN::normalizedLookAt() {

    float Egocent_x = LockOnTarget->position.x - position.x;
    float Egocent_y = LockOnTarget->position.y - position.y;

    float Angle_reltoX = atan2(Egocent_y, Egocent_x);
    float Difference = 2 * Params::pi - angle - Angle_reltoX;

    float wrappedDifference = wrapRange(Difference, -
     Params::pi, Params::pi);

    return normalize(wrappedDifference, -Params::pi,
     Params::pi);
}
```

It first determined relative x and y coordinates from the difference between the two objects, so that the function's result was only affected by their positions relative to each other, not based on the planes of the whole coordinate system.

Then the 'atan2()' function was used to determine the arc tangent to the relative point of the coordinate. After the difference had been calculated, this value was normalised to fit the range between zero and one. The result of this measurement showed 0.5 if the target was right in front of the RocketNN, values closer to zero indicated the target's position was more to the left, with values closer to 1 to the right respectively.

5.7 **'Interceptor'**

Interceptor was a class defining the properties of a rocket controlled by a neural net.

```
class Interceptor : public RocketNN {
public:
   Interceptor() {}
   Interceptor(sf::Vector2f position, float angle) :
    RocketNN(position, angle) {}
   virtual std::vector<float> getNNinputs() override;

   void reset();
   void reset(float angle);

};
```

5.7.1 Locally declared members

Eventually, the constructor of this class was called to construct an Interceptor object. The chain of constructors linking parameters started from this class, as it was an endpoint in hierarchy.

Sovereign constructor parameters:

- Position of the 'RocketHC' defined in a vector of 2 floating point numbers as its x and y coordinates respectively.
- Angle in radians representing the rotated angle of the object in respect to the plane of the graphical window.

Virtual functions:

- 'RocketNN:getNNinputs()' gets overridden. It is a virtual function of RocketNN because this way it is possible to later implement a neural net controlled Bandit against the Interceptor to learn against each other.

Local members:

- Resetting function declared both with and without the angle parameter.

5.7.2 Notable local function

This function gathered data which were later fed to the neural net as its inputs. All the calculations were done from relative coordinates and distances.

```
std::vector<float> Interceptor::getNNinputs() {

    std::vector<float> returnvector;

    float Egocent_x = LockOnTarget->position.x - position.x;
    float Egocent_y = LockOnTarget->position.y - position.y;

    float EgocentV_x = LockOnTarget->velocity.x - velocity.x;
    float EgocentV_y = LockOnTarget->velocity.y - velocity.y;


    // INPUT 1: Length of the velocity vector

    float velocityVectorLength = sqrt(pow(velocity.x, 2) +
     pow(velocity.y, 2));

    float normvelocityVectorLength = clamp(normalize(
     velocityVectorLength, 0, 20), 0, 1);

    returnvector.push_back(normvelocityVectorLength);
    // INPUT 2: Difference in direction between the face
     vector and velocity vector

    float VeloVec_xAxisDegree = atan2(velocity.y,
     velocity.x);

    float velocityFaceOffset = angle - VeloVec_xAxisDegree;
    returnvector.push_back(normalize(wrapRange(
     velocityFaceOffset, 0, 2 * Params::pi), 0, 2 *
     Params::pi));


    // INPUT 3: Target velocityvector's arc tangent to the
     x-axis

    float Enemyvelocity_xAxisDegree = atan2(EgocentV_y,
     EgocentV_x);
    float relEVxAD = angle - Enemyvelocity_xAxisDegree;
    float normEVxAD = normalize(wrapRange(relEVxAD, 0, 2 *
     Params::pi), 0, 2 * Params::pi);
    returnvector.push_back(normEVxAD);


    // INPUT 4 : Target's direction as values between
     Left-Centre-Right, compared to the interceptor's
     facevector

    returnvector.push_back(normalizedLookAt());


    // INPUT 5: Distance to the target

    returnvector.push_back(sqrt(pow(Egocent_x, 2) +
     pow(Egocent_y, 2)));

    return returnvector;
}
```

- Input 1 calculates the length of the Interceptor's velocity vector based on the Pythagoras theorem.
- Input 2 calculates the difference between face and velocity vectors by normalising the difference of angles between the arc tangent of the velocity vector and the Interceptor's own angle of rotation.
- Input 3 calculates the target's velocity vector's arc tangent to the x-axis.
- Input 4 uses the 'normalizedLookAt()' function described earlier, to calculate direction of the target compared to the Interceptor's face vector.
- Input 5 calculates the distance between the Interceptor and its target by using the Pythagoras theorem.

All this information is gathered to a vector of floating point numbers and later called to define the values of the neural net, updated each tick of the simulation with values dynamically.

6 MANAGER CLASSES

Manager classes were created to describe the flow and functionality of the program gathering together smaller functions.

The hierarchy of manager classes is seen on Figure 7.

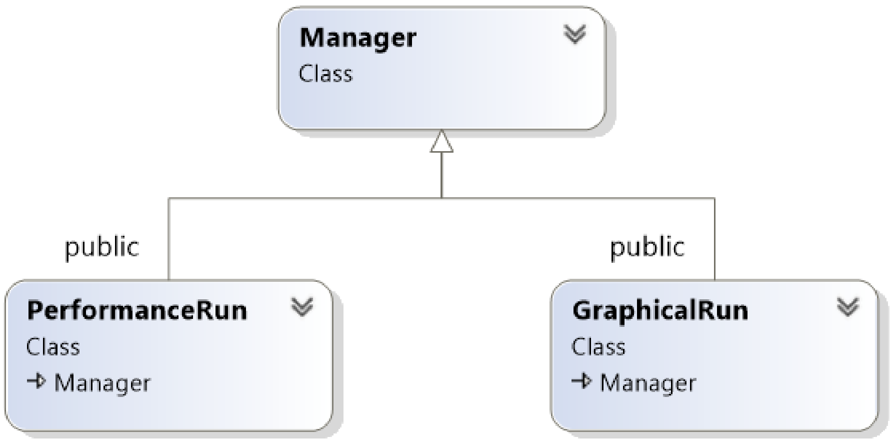


Figure 7 Diagram indicating the hierarchy of manager classes

'Manager' was the common abstract class of 'PerformanceRun' and 'GraphicalRun'. This meant that Manager was not directly called, but constructed through either instances of the derived class.

Both derived classes handled the same simulation, trained the neural net and the genetic evolution algorithm. Since 'GraphicalRun' rendered graphical objects to the screen it was not suitable to teach the AI as it would have taken an unnecessarily great deal of time due to graphical real-time rendering. For teaching purposes, 'PerformanceRun' was established to simulate each round of simulation as fast as the computer could handle these.

6.1 'Manager' base class

This class described all the common functions and objects which the derived classes both handled.

```
class Manager {
public:
    Manager() {

    I1 = Interceptor(Params::posRocketNN,
     Params::angleRocketNN);
    POP1 = Population();
    NN1 = NeuralNet();

    B1 = Bandit(Params::posRocketOPP,
     Params::angleRocketOPP);

    TargetB1 = Target();

    I1.DefineTarget(&B1);
    B1.DefineTarget(&TargetB1);
    }
    virtual bool Simulate(float angle) = 0;

    void Run();
    void Save(std::ostream& out, const sGenome& genome);
    void Load(std::istream& in, sGenome& genome);
    void SaveAll();
    void LoadAll();
    void SaveBestGenome();
    sGenome LoadBestGenome();

    void ManageTopGenomes(int topN);
    std::vector<sGenome> LoadTopGenomes();

    Interceptor I1;
    Population POP1;
    NeuralNet NN1;

    Bandit B1;

    Target TargetB1;

    int iGeneration = 1;
    int iGenome;
    float SimuTimeInSec = 0.0f;
};
```

This class declaration constructs an instance of the 'Interceptor' class named 'I1', as well as instances of 'Population' and 'NeuralNet' all labelled by the common number, indicating that they belong together. This is the way multiple objects should be added for later testing.

The Bandit rocket 'B1' also gets constructed. As of the current version of the program, the Bandit is not given AI controlls, even though this is easily implementable and have been experimented with. In case such thing is desired, sovereign neural net and population objects should be defined here for B1.

The Manager class is capable executing save and load features, as it is discussed in under the author's Serialization chapter.

It is also this class' job to handle counting the current index of generations and genomes.

6.1.1 'Run()' function

This function is acting like the frame around each individual simulation of the program.

```
void Manager::Run() {
    bool isEnded = false;

    for (iGeneration = iGeneration; !isEnded; ++iGeneration){

        TargetB1.reset();
    for (iGenome = 0; iGenome < Params::PopulationSize &&
     !isEnded; ++iGenome) {

        I1.reset();
        B1.reset(iGeneration);

        POP1.Genomes[iGenome].initposTargetB1 =
         TargetB1.position;

        POP1.Genomes[iGenome].initposI1 = I1.position;

        POP1.Genomes[iGenome].initposB1 = B1.position;

        float SimuAngle;

       for (float i = 0; i < Params::FitnessResolution; ++i){

           SimuAngle = i / (Params::FitnessResolution / 2) *
            Params::pi;
           isEnded = Simulate(SimuAngle);

           I1.reset(SimuAngle);
           B1.reset(iGeneration);
       }

       ManageTopGenomes(10);
    }

    std::cout << "Generation " << iGeneration << " ";
    std::cout << "Avg fitness: " <<
     POP1.CalculateAverageFitness();

    std::cout << " Best fitness: " <<
     POP1.getBestFitness() << std::endl;

    SaveBestGenome();

    POP1.Evolve();

    SaveAll();
    }
}
```

The first 'for()' cycle iterates between generations, with the nested second 'for()' cycle iterating between genomes all representing different simulation scenarios with different set of weights given to the neural net by the genetic algorithm.

Inside this second cycle, each simulation starts by resetting objects involved in the simulation.

The third 'for()' cycle constrained by 'Params::FitnessResolution' tests one genome to different simulation scenarios, as one lucky scenario alone is not deterministic enough to determine the usefulness of a genome by a greater fitness value assigned.

'ManageTopGenomes()' then updates the top saved genomes, as this is later described in the author's Serialization chapter.

After derived classes are done testing the last genome of each generation, information about the latest genome's performance in fitness scores is getting streamed to the console window. Finally, the genetic population's 'Evolve()' member function is called to guide the genomes into their next generation, saving the current state of last fully tested generation by calling the 'SaveAll()' function described under the author's Serialization paragraph.

6.2 Performance Run

The 'PerformanceRun' class has only one member besides its constructor, which overrides the virtual function 'Simulate()'

```
class PerformanceRun : public Manager {
public:
    PerformanceRun() {}

    virtual bool Simulate(float angle) override;
};
```

The 'Simulate()' function runs one simulation from until either of the rockets have gone out of bounds, or a target has been shot.

```
bool PerformanceRun::Simulate(float angle) {

    I1.angle = angle;

    int NrOfUpdates = 0;

    NN1.feedWeights(POP1.Genomes[iGenome].value);

    while (true) {

        if (I1.Collided || B1.Collided) { break; }

        if (I1.isOOB() && B1.isOOB()) { break; }

        if (NrOfUpdates / Params::PhysicsTimeStepsPerSecond
         >= Params::MaxSimulationTime) { break; }

    I1.SetNNControls(&NN1);
    I1.update();

    B1.update();

    TargetB1.update();

    NrOfUpdates++;
    }

    float SimulationTime = NrOfUpdates /
     Params::PhysicsTimeStepsPerSecond;

    if (POP1.Genomes[iGenome].fitness == 0)
        POP1.Genomes[iGenome].fitness =
         I1.calcFitness(SimuTimeInSec);
    else
        POP1.Genomes[iGenome].fitness +=
          I1.calcFitness(SimuTimeInSec);

    return false;
}
```

The fitness of a genome is dependent on its simulation time and since the 'PerformanceRun::Simulate()' function is not running real-time, it must calculate virtual time by accumulating the number of updates done. The variable 'NrOfUpdates' was declared here to handle this. To get the simulation's time in seconds, it has to be divided by 'Params::PhysicsTimeStepsPerSeconds' which is the same time constraint used for frame rate stabilizing of 'GraphicalRun::Simulate()'.

Each tick or each iteration of its main cycle, the 'SetNNControls(* NeuralNet)' function is called on the AI controlled rocket's instance, to update the inputs of its neural net in order to retrieve a new set of control command for the next tick of the simulation. The other objects are getting updated here from tick to tick as well.

A simulation's last step is to assign the fitness score to the genome, which was responsible for the weights of the neural net.

```
float RocketNN::calcFitness(float SimulationTime) {

    if (Collided)
        return normalize(Params::MaxSimulationTime –
         SimulationTime, 0, Params::MaxSimulationTime) + 2;
    else
        return normalize(1400 - ClosestDistanceToTarget, 0,
         1400);
}
```

The method of fitness score calculation first examines the 'Object::Collided' Boolean variable of the Interceptor, giving it a higher value in case it managed to collide with its target, where a faster interception means a higher score.

If that is not the case, the fitness score will simply be depending on the closest achieved distance to the target.

6.3 Graphical Run

The 'GraphicalRun' class has extended functions compared to 'PerformanceRun', which are used for visual representation, window handling, handling user keyboard inputs, changing speed, storing textures.

```
class GraphicalRun : public Manager {
public:
    GraphicalRun() :
     Window(sf::VideoMode(Params::WindowWidth,
     Params::WindowHeight), "Graphical Simulation ::
     Martin Pinter - HAMK Thesis",
     sf::Style::None), SpeedFactor(1.0f),
     Tstr(Params::BackgroundTexture) {
        BGT.loadFromFile(Tstr);
        BGT.setSmooth(false);
        BGT.setRepeated(false);
    }

    virtual bool Simulate(float angle) override;

    sf::Texture BGT;
    std::string Tstr;
    void drawBG();

    sf::RenderWindow Window;

    void ReplayBestGenome();
    void TopGenomes();

    void HandleUserInput();

    float SpeedFactor;

    bool abort = false;

    bool pressedKeys[sf::Keyboard::KeyCount] = { false };
};
```

The constructor renders a window object and sets the texture of the background, by its location stored in 'Params'.

'ReplayBestGenome()' and 'TopGenomes()' are replay functions implemented to show the best solutions of the artificial intelligence.

Keypress event of all keyboard keys are set to false here. This helps avoiding unwanted results, when a key is being held on constructing the class.

```
bool GraphicalRun::Simulate(float angle) {

    I1.angle = angle;
    sf::Clock Clock;
    SimuTimeInSec = 0.0f;
    float PhysicsTimeStepAccumulator = 0.0f;

    NN1.feedWeights(POP1.Genomes[iGenome].value);

    while (Window.isOpen()) {

        if (I1.Collided || B1.Collided)
            break;
        if (I1.isOOB() && B1.isOOB())
            break;

        const sf::Time FrameTime = Clock.restart();
        float FrameSeconds = FrameTime.asSeconds();
        if (FrameSeconds > 0.1f) FrameSeconds = 0.1f;
        PhysicsTimeStepAccumulator += FrameSeconds;

        HandleUserInput();

        while (PhysicsTimeStepAccumulator >= SpeedFactor *
         Params::PhysicsTimeStep) {
            I1.SetNNControls(&NN1);
            I1.update();
            B1.update();
            TargetB1.update();

            PhysicsTimeStepAccumulator -= SpeedFactor *
             Params::PhysicsTimeStep;
            SimuTimeInSec += Params::PhysicsTimeStep;
        }

        Window.clear(sf::Color::Black);

        drawBG();
        I1.draw(Window);
        B1.draw(Window);
        TargetB1.draw(Window);
        GUI(Window, iGeneration, iGenome);

        Window.display();
    }

if (!abort) {
    if (POP1.Genomes[iGenome].fitness == 0)

        POP1.Genomes[iGenome].fitness =
         I1.calcFitness(SimuTimeInSec);
    else
        POP1.Genomes[iGenome].fitness +=
         I1.calcFitness(SimuTimeInSec);
}

abort = false;

std::cout << "Fitness: " << POP1.Genomes[iGenome].fitness <<
 std::endl;

return !Window.isOpen();
}
```

The 'GraphicalRun::Simulate()' is a similar function to the 'Performance::Simulate()', but extended by visual rendering and framerate calculation.

After starting a clock for measuring the simulation's time, the neural net requests a genome to be fed into the net's weights. The genetic algorithm supplies this, and so the simulation begins.

A stable framerate is essential for deterministic fitness results. In order to guarantee this, a manual framerate calculation method has been implemented. This method does not allow the objects to be updated more often, than it is described in 'Params'.

For the rendering of each frame, the followings sequence has to be implemented as it represents general handling of frames:

– 'clear(sf::Color)' clears the window from any previous content, resetting it with the color given as its parameter.

– 'draw(sf::RenderWindow&)' function must be called on all objects meant to be visualized. Note that the Graphical User Interface (GUI) object has its 'draw()' function called in its constructor.
– 'display()' function displays all the objects which called their 'draw()' in the previous step, thus ending the frame.

This function returns the state of the window negated, as 'Manager::Run()' expects to be informed once the window has been closed.

7 SERIALIZATION

Serialization is a method for saving and loading data of determined syntax. Since C++ does not offer direct serialization methods, the author has implemented them manually in the Manager classes.

7.1 Saving a single genome

A single genome is saved using the 'Manager::Save(std::ostream&, const sGenome&)' function.

```
void Manager::Save(std::ostream& out, const sGenome& genome){

    out << genome.fitness << std::endl;

    out << genome.value.size() << std::endl;

    for (float value : genome.value) {
        out << value << ' ';
    }

    out << std::endl;
    out << genome.initposI1.x << ' ' <<
     genome.initposI1.y << ' ';
    out << genome.initposB1.x << ' ' <<
     genome.initposB1.y << ' ';
    out << genome.initposTargetB1.x << ' ' <<
     genome.initposTargetB1.y << std::endl;
}
```

This function receives an output stream reference, and a constant genome reference. References mean that on the contrary to regular parameterizing, not a duplicate, but the same instance of the object is being passed.

The '<<' and '>>' are output and input stream operators respectively.

Values saved in order:
– Fitness of the genome.
– Size of the genome (number of weights allocated).
– Values of each genome.
– Initial positions of objects for replay mode.

7.2 Saving all genomes

The 'SaveAll()' function utilizes the 'Save()' function explained above.

```
void Manager::SaveAll() {

    std::ofstream out("SaveFile.txt");

    out << iGeneration << std::endl;

    out << iGenome << std::endl;

    out << POP1.Genomes.size() << std::endl;

    for (const sGenome& genome : POP1.Genomes) {
        Save(out, genome);
    }

}
```

Since the previous 'Save()' function only implements saving one genome, the 'SaveAll()' function handles exporting the complete previous population with its corresponding attributes.

After defining an output file stream pointing to the location of the desired file, it exports the following attributes via the 'std::ofstream':
- The current generation's index.
- The current genome's number.
- The population's size (number of genomes within).
- The 'Save()' function on each genome of the population, described into above.

7.3 Loading a single genome

If serialized properly, the 'Load()' function executes the same attributes in the same order as it was defined within the 'Save()' function.

```
void Manager::Load(std::istream& in, sGenome& genome) {

    in >> genome.fitness;

    int size;
    in >> size;

    genome.value.resize(size);

    for (float& value : genome.value) {
        in >> value;
    }

    in >> genome.initposI1.x >> genome.initposI1.y;
    in >> genome.initposB1.x >> genome.initposB1.y;
    in >> genome.initposTargetB1.x >>
     genome.initposTargetB1.y;
}
```

Note how this function is symmetrical to the one saving a single genome. It also has its corresponding 'LoadAll()' function symmetrical to the 'SaveAll()' function.

This way the neural net is able to pick up learning from where it left off the last time. The serialization methods implemented ensure continuous learning of the artificial intelligence with options to start a new learning course all over again, as defined in the main menu.

8 CONCLUSION

The author's aim was to explain the methodology of the development of artificial intelligence by familiarizing the reader with virtually inter-exchangeable concepts between the biological template of neural systems and the way they are artificially programmed to achieve similar results.

The purpose of code snippets appearing in the content of the code was to highlight the system's working principles. The full code is available in Appendices 4-36.

Developing this program was an exciting task, awarding experience and knowledge about software development to the author. Although the author as a student of Automation Engineering had previous knowledge about programming, this project was a huge increase in challenge and complexity. It is the author's aim to continue extending his knowledge about artificial intelligence studies and programming in the future.

The project was developed in Microsoft Visual Studio Community 2015.

There were many architectural changes to the code to reach its current state. Thanks to object oriented principles of development, the code was built in a way to be easily extended and restored later. A GitHub repository containing the complete functional project with documentation and changes made to it is available at: https://github.com/martindpinter/Bachelors-Thesis.

SOURCES

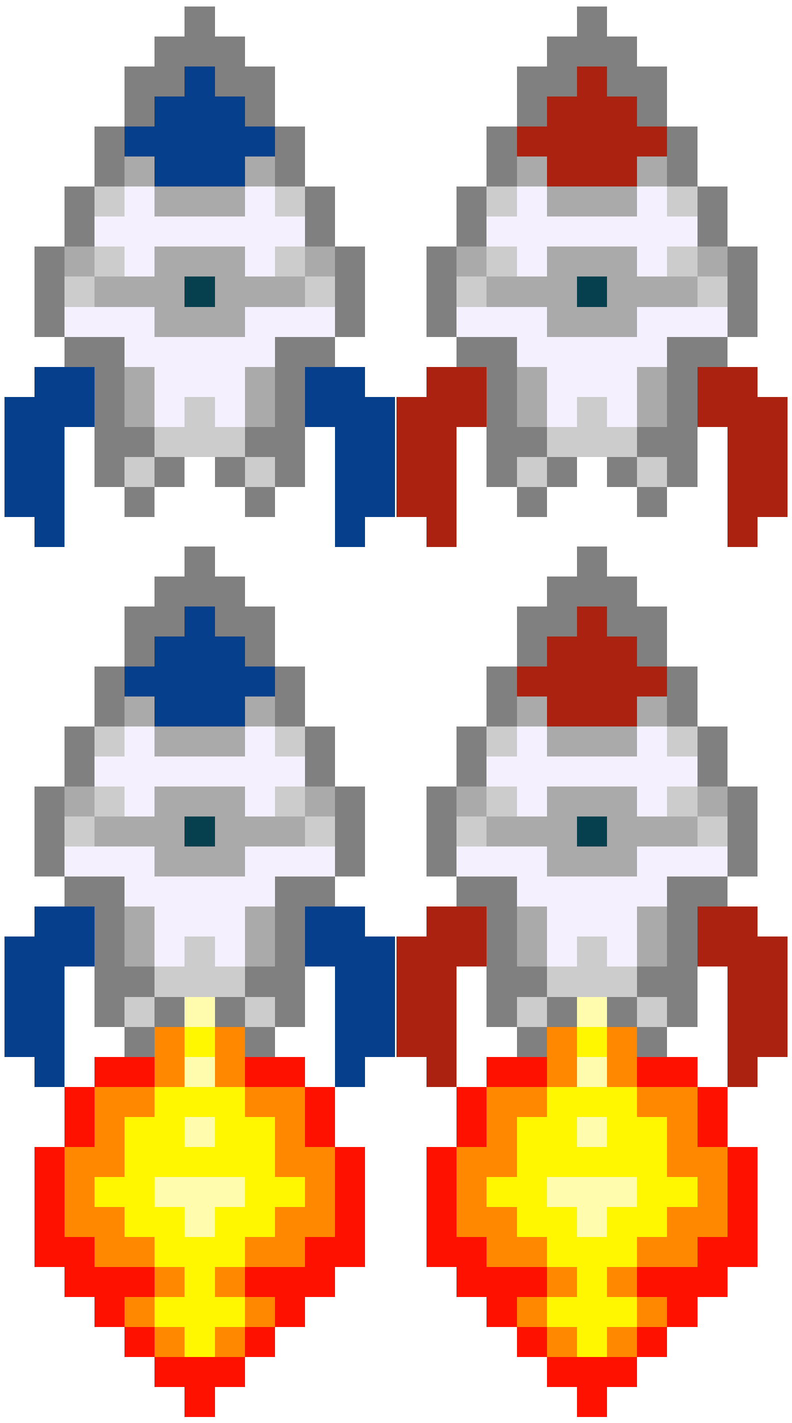
Cplusplus.com 2016, accessed on 13.5.2016,
<http://www.cplusplus.com>

Buckland, M 2016, AI-Junkie, accessed on 05.10.2016,
<http://ai-junkie.com>

Simple and Fast Multimedia Library (SFML), accessed on 03.12.2016,
<http://www.sfml-dev.org>

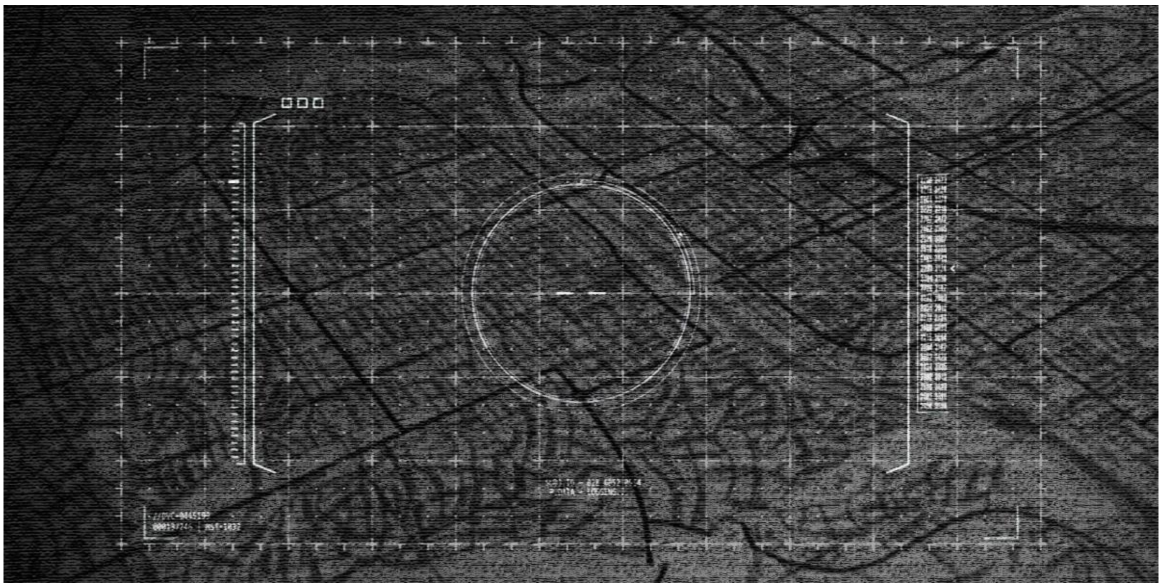
Appendix 1

ROCKET TEXTURES



Appendix 2

BACKGROUND TEXTURE



Appendix 3

TARGET TEXTURE

Appendix 4

Contents of the file 'main.cpp'

```
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>

#include "PerformanceRun.h"
#include "GraphicalRun.h"
#include "Sandbox.h"

#define PI 3.141592;


int main() {

    srand(time(nullptr));

    std::cout << "MENU" << std::endl;
    std::cout << "1. Start New Performance Teaching" <<
     std::endl;
    std::cout << "2. Continue Performance Teaching" <<
     std::endl;
    std::cout << "3. Start New Graphical Simulation" <<
     std::endl;
    std::cout << "4. Continue Graphical Simulation" <<
     std::endl;
    std::cout << "5. Sandbox Mode" << std::endl;
    std::cout << "6. Replay Best Genome" << std::endl;
    std::cout << "7. Top Genomes" << std::endl;
    std::cout << "X. Quit Program" << std::endl;


    bool quit = false;

    while (!quit) {

        char menuChoice;
        std::cin >> menuChoice;

        if (menuChoice == '1') {
            PerformanceRun GM;
            GM.Run();
        }
        else if (menuChoice == '2') {
            PerformanceRun GM;
            GM.LoadAll();
            GM.Run();

        }
        else if (menuChoice == '3') {
            GraphicalRun GM;
            //GM.LoadAll();
            GM.Run();
        }
        else if (menuChoice == '4') {
            GraphicalRun GM;
            GM.LoadAll();
            GM.Run();
        }
        else if (menuChoice == '5') {
            Sandbox GM;
            GM.Run();
        }
        else if (menuChoice == '6') {
            GraphicalRun GM;
            GM.ReplayBestGenome();
            //char endchar;
            //std::cin >> endchar;
        }
        else if (menuChoice == '7') {
            GraphicalRun GM;
            GM.TopGenomes();
        }
        else if (menuChoice == 'x') {
            quit = true;
        }
    }

    return 0;
}
```

Appendix 5

Contents of the file 'Bandit.h'

```
#pragma once
#include "RocketController.h"

class Bandit : public RocketController {
public:

    Bandit() {}
    Bandit(sf::Vector2f position, float angle) :
     RocketController(position, angle, Params::RedNTT,
     Params::RedFTT) {}

    void reset(int seed);
    virtual void controls() override;

    virtual void update() override;
};
```

Appendix 6

Contents of the file 'Bandit.cpp'

```
#include "Bandit.h"
#include <random>
void Bandit::reset(int seed) {

    std::mt19937 gen(seed);
    std::uniform_int_distribution<> disX(
     Params::WindowWidth * 0.9, Params::WindowWidth * 0.9 +
     Params::WindowWidth / 10);
    std::uniform_int_distribution<>
     disY(Params::WindowHeight * 0.1, Params::WindowHeight *
     0.1 + Params::WindowHeight * 0.8);

    int randX = disX(gen);
    int randY = disY(gen);

    position = sf::Vector2f(randX, randY);
    velocity = Params::nullvec;

    float Egocent_x = LockOnTarget->position.x - position.x;
```

```
        float Egocent_y = LockOnTarget->position.y - position.y;
        angle = atan2(Egocent_y, Egocent_x);

        throttle = 0.0f;
        angular_throttle = 0.0f;
        angular_velocity = 0.0f;
        Collided = false;
    }

    void Bandit::controls() {
        accelerate(0.01f);
    }

    void Bandit::update() {

        RocketController::update();
        TargetHit();
    }
```

Appendix 7

Contents of the file 'GraphicalRun.h'

```
#pragma once
#include "Manager.h"

class GraphicalRun : public Manager {
public:
    GraphicalRun() :
        Window(sf::VideoMode(Params::WindowWidth,
        Params::WindowHeight), "Graphical Simulation ::
        Martin Pinter - HAMK Thesis", sf::Style::None),
        SpeedFactor(1.0f), Tstr(Params::BackgroundTexture) {

            BGT.loadFromFile(Tstr);
            BGT.setSmooth(false);
            BGT.setRepeated(false);
        }

    sf::Texture BGT;
    std::string Tstr;
    void drawBG();

    sf::RenderWindow Window;

    virtual bool Simulate(float angle) override;

    void ReplayBestGenome();
    void TopGenomes();

    void HandleUserInput();

    float SpeedFactor;

    bool abort = false;

    bool pressedKeys[sf::Keyboard::KeyCount] = { false };
};
```

Appendix 8

Contents of the file 'GraphicalRun.cpp'

```
#include "GraphicalRun.h"
#include "GUI.h"
#include "Utilities.h"
#include "Target.h"

bool GraphicalRun::Simulate(float angle) {

    I1.angle = angle;

    sf::Clock Clock;

    SimuTimeInSec = 0.0f;

    float PhysicsTimeStepAccumulator = 0.0f;

    NN1.feedWeights(POP1.Genomes[iGenome].value);

    while (Window.isOpen()) {

        if (I1.Collided || B1.Collided)
            break;

        if (I1.isOOB() && B1.isOOB())
            break;

        const sf::Time FrameTime = Clock.restart();
        float FrameSeconds = FrameTime.asSeconds();
        if (FrameSeconds > 0.1f) FrameSeconds = 0.1f;
        PhysicsTimeStepAccumulator += FrameSeconds;

        HandleUserInput();

        while (PhysicsTimeStepAccumulator >= SpeedFactor *
         Params::PhysicsTimeStep) {

            I1.SetNNControls(&NN1);
            I1.update();
            B1.update();
            TargetB1.update();

            PhysicsTimeStepAccumulator -= SpeedFactor *
             Params::PhysicsTimeStep;
            SimuTimeInSec += Params::PhysicsTimeStep;
        }


        Window.clear(sf::Color::Black);

        drawBG();
        I1.draw(Window);
        B1.draw(Window);
        TargetB1.draw(Window);

        GUI(Window, iGeneration, iGenome);

        Window.display();
    }
    if (!abort) {
        if (POP1.Genomes[iGenome].fitness == 0)
            POP1.Genomes[iGenome].fitness =
             I1.calcFitness(SimuTimeInSec);
        else
            POP1.Genomes[iGenome].fitness +=
            I1.calcFitness(SimuTimeInSec);
    }
    abort = false;

    std::cout << "Fitness: " << POP1.Genomes[iGenome].fitness
     << std::endl;

    return !Window.isOpen();
}



void GraphicalRun::TopGenomes() {
    sf::Clock Clock;

    SimuTimeInSec = 0.0f;

    float PhysicsTimeStepAccumulator = 0.0f;

    std::vector<sGenome> TopGenomes = LoadTopGenomes();

    for (sGenome& genome : TopGenomes) {
        NN1.feedWeights(genome.value);

        I1.reset();
        B1.reset(8);
        TargetB1.reset();

        I1.position = genome.initposI1;
        B1.position = genome.initposB1;
        TargetB1.position = genome.initposTargetB1;

        float Egocent_x = TargetB1.position.x -
         B1.position.x;
        float Egocent_y = TargetB1.position.y -
         B1.position.y;
        B1.angle = atan2(Egocent_y, Egocent_x);

        while (Window.isOpen()) {

            if (I1.Collided || B1.Collided)
                break;

            if (I1.isOOB() && B1.isOOB())
                break;

            const sf::Time FrameTime = Clock.restart();
            float FrameSeconds = FrameTime.asSeconds();
            if (FrameSeconds > 0.1f) FrameSeconds = 0.1f;
            PhysicsTimeStepAccumulator += FrameSeconds;

            while (PhysicsTimeStepAccumulator >= SpeedFactor
             * Params::PhysicsTimeStep) {

                I1.SetNNControls(&NN1);
                I1.update();
                B1.update();
                TargetB1.update();

                PhysicsTimeStepAccumulator -= SpeedFactor *
                 Params::PhysicsTimeStep;
                SimuTimeInSec += Params::PhysicsTimeStep;
            }
```

```
                Window.clear(sf::Color::Black);

                drawBG();
                I1.draw(Window);
                B1.draw(Window);
                TargetB1.draw(Window);

                GUI(Window, iGeneration, iGenome);

                Window.display();
            }
        }
    }

    void GraphicalRun::ReplayBestGenome() {
        sf::Clock Clock;

        SimuTimeInSec = 0.0f;

        float PhysicsTimeStepAccumulator = 0.0f;

        sGenome BestGenome = LoadBestGenome();

        NN1.feedWeights(BestGenome.value);

        I1.position = BestGenome.initposI1;
        B1.position = BestGenome.initposB1;
        TargetB1.position = BestGenome.initposTargetB1;

        float Egocent_x = TargetB1.position.x - B1.position.x;
        float Egocent_y = TargetB1.position.y - B1.position.y;
        B1.angle = atan2(Egocent_y, Egocent_x);

        while (Window.isOpen()) {
            if (I1.Collided || B1.Collided)
                break;

            if (I1.isOOB() && B1.isOOB())
                break;

            const sf::Time FrameTime = Clock.restart();
            float FrameSeconds = FrameTime.asSeconds();
            if (FrameSeconds > 0.1f) FrameSeconds = 0.1f;
            PhysicsTimeStepAccumulator += FrameSeconds;

            while (PhysicsTimeStepAccumulator >= SpeedFactor *
             Params::PhysicsTimeStep) {

                I1.SetNNControls(&NN1);
                I1.update();
                B1.update();
                TargetB1.update();

                PhysicsTimeStepAccumulator -= SpeedFactor *
                 Params::PhysicsTimeStep;
                SimuTimeInSec += Params::PhysicsTimeStep;
            }


            Window.clear(sf::Color::Black);
            drawBG();
            I1.draw(Window);
            B1.draw(Window);
            TargetB1.draw(Window);

            GUI(Window, iGeneration, iGenome);

            Window.display();
        }
    }



    void GraphicalRun::HandleUserInput() {

        sf::Event event;

        while (Window.pollEvent(event)) {

            switch (event.type) {

            case sf::Event::Closed:
                Window.close();
                break;
            case sf::Event::KeyPressed:

                pressedKeys[event.key.code] = true;

                switch (event.key.code) {
                case sf::Keyboard::S:
                    SaveAll();
                    break;
                case sf::Keyboard::L:
                    I1.reset();
                    B1.reset(iGeneration);
                    LoadAll();
                    abort = true;
                    break;
                case sf::Keyboard::Add:
                    SpeedFactor *= 1.1f;
                    break;
                case sf::Keyboard::Subtract:
                    SpeedFactor /= 1.1f;
                    break;
                case sf::Keyboard::Escape:
                    Window.close();
                    break;
                }
                break;
            }
        }
    }

    void GraphicalRun::drawBG() {
        sf::Texture* BGpointer;
        BGpointer = &BGT;

        sf::Sprite BGSprite;
        BGSprite.setTexture(*BGpointer);
        BGSprite.setPosition(0, 0);

        Window.draw(BGSprite);
    }
```

Appendix 9

Contents of the file GUI.h'

```
#pragma once
#include <SFML/Graphics.hpp>

class GUI {
public:
    GUI(sf::RenderWindow& Window, int Generation, int Genome)
: Generation(Generation), Genome(Genome) { draw(Window);  }

    int Generation;
    int Genome;

    void draw(sf::RenderWindow& Window);
};
```

Appendix 10

Contents of the file GUI.cpp'

```
#include "GUI.h"
#include "Params.h"

void GUI::draw(sf::RenderWindow& Window) {

    sf::Font font;
    sf::Text text;
    std::string GUIString;

    font.loadFromFile("font.ttf");

    text.setFont(font);
    text.setCharacterSize(15);
    text.setColor(sf::Color::White);

    GUIString = "Generation: ";
    GUIString += std::to_string(Generation);
    GUIString += "/";
    GUIString += std::to_string(Params::MaxGenerations);

    GUIString += "\nGenome: ";
    GUIString += std::to_string(Genome);
    GUIString += "/";
    GUIString += std::to_string(Params::PopulationSize);

    GUIString += "\n AvgFit: ";

    text.setString(GUIString);
    text.setPosition(sf::Vector2f(10, 10));

    Window.draw(text);

}
```

Appendix 11

Contents of the file 'Interceptor.h'

```
#pragma once
#include "RocketNN.h"
```

```
class Interceptor : public RocketNN {
public:
    Interceptor() {}
    Interceptor(sf::Vector2f position, float angle) :
     RocketNN(position, angle) {}

    virtual std::vector<float> getNNinputs() override;

    void reset();
    void reset(float angle);
};
```

Appendix 12

Contents of the file 'Interceptor.cpp'

```
#include "Interceptor.h"

void Interceptor::reset() {

    position = Params::posRocketNN;
    velocity = Params::nullvec;
    //angle = Params::angleRocketNN;
    int angleConst = randfloat(1.0, 2.5);
    angle = angleConst * Params::pi;
    throttle = 0.0f;
    angular_throttle = 0.0f;
    angular_velocity = 0.0f;
    rotationalSum = 0.0f;
    Collided = false;
    LookAtScore = 0;

}
void Interceptor::reset(float inputangle) {

    Interceptor::reset();
    angle = inputangle;
}

std::vector<float> Interceptor::getNNinputs() {

    std::vector<float> returnvector;

    float Egocent_x = LockOnTarget->position.x - position.x;
    float Egocent_y = LockOnTarget->position.y - position.y;

    float EgocentV_x = LockOnTarget->velocity.x - velocity.x;
    float EgocentV_y = LockOnTarget->velocity.y - velocity.y;


    // INPUT 1: Length of the velocity vector
    float velocityVectorLength = sqrt(pow(velocity.x, 2) +
     pow(velocity.y, 2));
    float normvelocityVectorLength =
    clamp(normalize(velocityVectorLength, 0, 20), 0, 1);
    returnvector.push_back(normvelocityVectorLength);


    // INPUT 2: Difference in direction between the face
     vector and velocity vector
    float VeloVec_xAxisDegree = atan2(velocity.y,
     velocity.x);
    float velocityFaceOffset = angle - VeloVec_xAxisDegree;
    returnvector.push_back(normalize(wrapRange(
     velocityFaceOffset,  0,  2  *  Params::pi),  0,  2  *
Params::pi));


    // INPUT 3: Target velocityvector's arc tangent to the
     x-axis
    float Enemyvelocity_xAxisDegree = atan2(EgocentV_y,
     EgocentV_x);
    float relEVxAD = angle - Enemyvelocity_xAxisDegree;
    float normEVxAD = normalize(wrapRange(relEVxAD, 0, 2 *
     Params::pi), 0, 2 * Params::pi);
    returnvector.push_back(normEVxAD);


    // INPUT 3 : Target's direction as values between Left-
     Centre-Right, compared to the interceptor's facevector
    returnvector.push_back(normalizedLookAt());

    // INPUT 4: Distance to the target
    returnvector.push_back(sqrt(pow(Egocent_x, 2) +
     pow(Egocent_y, 2)));

    return returnvector;
}
```

Appendix 13

Contents of the file 'Manager.h'

```
#pragma once
#include <iostream>
#include "NeuralNet.h"
#include "Interceptor.h"
#include "Bandit.h"
#include "Target.h"


class Manager {
public:
    Manager() {

        I1 = Interceptor(Params::posRocketNN,
         Params::angleRocketNN);
        POP1 = Population();
        NN1 = NeuralNet();

        B1 = Bandit(Params::posRocketOPP,
         Params::angleRocketOPP);

        TargetB1 = Target();

        I1.DefineTarget(&B1);
        B1.DefineTarget(&TargetB1);
    }

    virtual bool Simulate(float angle) = 0;

    void Run();
    void Save(std::ostream& out, const sGenome& genome);
    void Load(std::istream& in, sGenome& genome);
    void SaveAll();
    void LoadAll();
    void SaveBestGenome();
```

```
        sGenome LoadBestGenome();

        void ManageTopGenomes(int topN);
        std::vector<sGenome> LoadTopGenomes();

        Interceptor I1;
        Population POP1;
        NeuralNet NN1;

        Bandit B1;

        Target TargetB1;

        int iGeneration = 1;
        int iGenome;
        float SimuTimeInSec = 0.0f;
    };
```

Appendix 14

Contents of the file 'Manager.cpp'

```
#include <fstream>
#include "Manager.h"
#include "Params.h"
#include <fstream>
#include <iomanip>

void Manager::Run() {
    bool isEnded = false;

   for (iGeneration = iGeneration; !isEnded; ++iGeneration){
        TargetB1.reset();


        for (iGenome = 0; iGenome < Params::PopulationSize &&
         !isEnded; ++iGenome) {

            break;

            I1.reset();
            B1.reset(iGeneration);

            POP1.Genomes[iGenome].initposTargetB1 =
             TargetB1.position;
            POP1.Genomes[iGenome].initposI1 = I1.position;
            POP1.Genomes[iGenome].initposB1 = B1.position;

            float SimuAngle;

            for (float i = 0; i < Params::FitnessResolution;
             ++i) {
                SimuAngle = i / (Params::FitnessResolution /
                 2) * Params::pi;
                isEnded = Simulate(SimuAngle);
                I1.reset(SimuAngle);
                B1.reset(iGeneration);
            }


            ManageTopGenomes(10);
        }
        std::cout << "Generation " << iGeneration << " ";
```

```
            std::cout << "Avg fitness: " <<
             POP1.CalculateAverageFitness();
            std::cout << " Best fitness: " <<
             POP1.getBestFitness() << std::endl;

            SaveBestGenome();

            POP1.Evolve();

            SaveAll();

        }
    }

    void Manager::SaveAll() {

        std::ofstream out("SaveFile.txt");

        out << iGeneration << std::endl;

        out << iGenome << std::endl;

        out << POP1.Genomes.size() << std::endl;

        for (const sGenome& genome : POP1.Genomes) {
            Save(out, genome);
        }

    }

    void Manager::ManageTopGenomes(int topN) {
        if (POP1.Genomes[iGenome].fitness > 1) {
            std::vector<sGenome> TopGenomes;
            std::ifstream in("TopGenomes.txt");

            int size = 0;
            in >> size;

            if (size > 0) {
                TopGenomes.resize(size);

                for (sGenome& genome : TopGenomes) {
                    Load(in, genome);
                }
            }
            in.close();


            TopGenomes.push_back(POP1.Genomes[iGenome]);

            std::sort(TopGenomes.begin(), TopGenomes.end(),
             [](const sGenome& lhs, const sGenome& rhs) {
                return lhs.fitness > rhs.fitness;    });

            while (TopGenomes.size() > topN)
                TopGenomes.pop_back();

            std::ofstream out("TopGenomes.txt");

            size = TopGenomes.size();

            out << size++ << std::endl;
            for (sGenome& genome : TopGenomes) {
                Save(out, genome);
            }
        }

    }

    std::vector<sGenome> Manager::LoadTopGenomes() {
        std::vector<sGenome> returnvector;
        std::ifstream in("TopGenomes.txt");

        int size = 0;
        in >> size;

        if (size > 0) {
            returnvector.resize(size);

            for (sGenome& genome : returnvector) {
                Load(in, genome);
            }
        }
        return returnvector;

    }

    void Manager::SaveBestGenome() {

        sGenome & BestGenome =
         *std::max_element(POP1.Genomes.begin(),
         POP1.Genomes.end(), [](const sGenome& lhs,
         const sGenome& rhs) { return lhs.fitness < rhs.fitness;
    });

        if (BestGenome.fitness > LoadBestGenome().fitness) {

            std::ofstream out("BestGenome.txt");

            out << BestGenome.fitness << std::endl;

            out << BestGenome.value.size() << std::endl;

            for (float value : BestGenome.value) {
                out << value << ' ';
            }

            out << std::endl;
            out << BestGenome.initposI1.x << ' ' <<
             BestGenome.initposI1.y << ' ';
            out << BestGenome.initposB1.x << ' ' <<
             BestGenome.initposB1.y << ' ';
            out << BestGenome.initposTargetB1.x << ' ' <<
             BestGenome.initposTargetB1.y;

        }
    }

    sGenome Manager::LoadBestGenome() {
        std::ifstream in("BestGenome.txt");
        sGenome BestGenome;
        Load(in, BestGenome);

        /*std::cout << BestGenome.fitness << std::endl;
        std::cout << BestGenome.value.size() << std::endl;
        for (float value : BestGenome.value) {
```

```
        std::cout << value << " ";
    }*/

    in >> BestGenome.initposI1.x >> BestGenome.initposI1.y;
    in >> BestGenome.initposB1.x >> BestGenome.initposB1.y;
    in >> BestGenome.initposTargetB1.x >>
     BestGenome.initposTargetB1.y;

    return BestGenome;
}


void Manager::Save(std::ostream& out, const sGenome& genome)
{

    out << genome.fitness << std::endl;

    out << genome.value.size() << std::endl;

    for (float value : genome.value) {
        out << value << ' ';
    }

    out << std::endl;
    out << genome.initposI1.x << ' ' << genome.initposI1.y <<
     ' ';
    out << genome.initposB1.x << ' ' << genome.initposB1.y <<
     ' ';
    out << genome.initposTargetB1.x << ' ' <<
     genome.initposTargetB1.y << std::endl;
}

void Manager::LoadAll() {
    std::vector<sGenome> newPopulation;
    std::ifstream in("SaveFile.txt");
    in >> iGeneration;

    in >> iGenome;
    if (iGenome >= Params::PopulationSize)
        iGenome = 0;

    int PopulationSize;
    in >> PopulationSize;
    newPopulation.resize(PopulationSize);

    int genomeCounter = 0;
    for (sGenome& genome : newPopulation) {
        if (genomeCounter < PopulationSize) {
            Load(in, genome);
            genomeCounter++;
        }
    }

    POP1.Genomes = newPopulation;
}

void Manager::Load(std::istream& in, sGenome& genome) {

    in >> genome.fitness;

    int size;
    in >> size;
    genome.value.resize(size);

    for (float& value : genome.value) {
        in >> value;
    }

    in >> genome.initposI1.x >> genome.initposI1.y;
    in >> genome.initposB1.x >> genome.initposB1.y;
    in >> genome.initposTargetB1.x >>
     genome.initposTargetB1.y;

}
```

Appendix 15

Contents of the file 'NeuralNet.h'

```
#pragma once
#include <vector>
#include "Utilities.h"
#include "Params.h"
#include "Population.h"



struct sNeuron {

    std::vector<float> weights;
    sNeuron(int nrOfInputs);
};


struct sNeuronLayer {

    int nrOfNeurons;
    std::vector<sNeuron> one_neuronLayer;

    sNeuronLayer(int nrOfNeurons, int nrOfInputsPerNeuron);
};

class NeuralNet {
private:
    std::vector<sNeuronLayer> all_neuronLayers;

public:

    NeuralNet() { createNet();  }

    void createNet();
    void feedWeights(std::vector<float> inputWeights);
    std::vector<float> evaluate(std::vector<float>);
};
```

Appendix 16

Contents of the file 'NeuralNet.cpp'

```
#include "NeuralNet.h"


sNeuron::sNeuron(int nrOfInputs) {
    for (int i = 0; i < nrOfInputs + 1; ++i) {    // +1 since
the bias
        weights.push_back(randClamped());
    }
}

sNeuronLayer::sNeuronLayer(int nr0fNeurons,
 int nr0fInputsPerNeuron) : nr0fNeurons(nr0fNeurons) {

    for (int i = 0; i < nr0fNeurons; ++i) {


one_neuronLayer.push_back(sNeuron(nr0fInputsPerNeuron));
    }
}


void NeuralNet::feedWeights(std::vector<float> inputWeights)
{
    if (inputWeights.size() != Params::WeightCount) {
        std::cout << "The number of input weights does not
         match the weight count." << std::endl;
    }
    else {
        int weightIndex = 0;
        for (int i = 0; i < all_neuronLayers.size(); ++i) {
            for (int j = 0; j <
             all_neuronLayers[i].one_neuronLayer.size();
             ++j) {
                for (int k = 0; k <
                 all_neuronLayers[i].one_neuronLayer[j].
                 weights.size(); ++k) {

all_neuronLayers[i].one_neuronLayer[j].weights[k] =
 inputWeights[weightIndex++];

                }

            }
        }
    }
}

void NeuralNet::createNet() {
    //Creates the structure of layers
    if (Params::nr0fHiddenLayers > 0) {
        for (int i = 0; i < Params::nr0fHiddenLayers; ++i) {

            if (i == 0)
                all_neuronLayers.push_back(sNeuronLayer(
                 Params::NeuronsPerHiddenLayer,
                 Params::nr0fInputs));
            else
                all_neuronLayers.push_back(sNeuronLayer(
                 Params::NeuronsPerHiddenLayer,
                 Params::NeuronsPerHiddenLayer));
        }
        all_neuronLayers.push_back(sNeuronLayer(
         Params::nr0fOutputs,
         Params::NeuronsPerHiddenLayer));
    }
    else {
        all_neuronLayers.push_back(sNeuronLayer(
         Params::nr0fOutputs, Params::nr0fInputs));
    }
}
```

```
std::vector<float>
NeuralNet::evaluate(std::vector<float> inputvector) {

    std::vector<float> OneLayerInputs = inputvector;
    std::vector<float> OneLayerOutputs;
    std::vector<float> OutputVector;

    // Evaluating through the Input Layer and Hidden Layers

    for (int i = 0; i < Params::nrOfHiddenLayers; ++i) {
        if (i > 0) {
            OneLayerInputs = OneLayerOutputs;
        }

        for (int j = 0; j <
         all_neuronLayers[i].one_neuronLayer.size(); ++j) {

            float total = 0.0f;

            for (int k = 0; k < all_neuronLayers[i].
             one_neuronLayer[j].weights.size(); ++k) {
                if (k != all_neuronLayers[i].
                 one_neuronLayer[j].weights.size() - 1) {
                   total +=
                    all_neuronLayers[i].one_neuronLayer[j].
                    weights[k] * OneLayerInputs[k];
                }
                else {
                    total +=
                     all_neuronLayers[i].one_neuronLayer[j]
                     .weights[k] * Params::BiasValue;
                }

            }
            OneLayerOutputs.push_back(Sigmoid(total));
        }
    }

    OneLayerInputs = OneLayerOutputs;

    // Evaluating through the Output layer

    for (int i = 0; i < Params::nrOfOutputs; ++i) {

        float total = 0.0f;

        for (int j = 0; j <
         all_neuronLayers.back().
         one_neuronLayer[i].weights.size(); ++j) {
            if (j != all_neuronLayers.back().
             one_neuronLayer[i].weights.size() - 1) {
                total += all_neuronLayers.back().
                 one_neuronLayer[i].weights[j] *
                 OneLayerOutputs[j];
            }
            else {
                total += all_neuronLayers.back().
                 one_neuronLayer[i].weights[j] *
                 Params::BiasValue;
            }
        }
```

```
            OutputVector.push_back(Sigmoid(total));
        }

        return OutputVector;
    }
```

Appendix 17

Contents of the file 'Object.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Params.h"

class Object {
public:

    Object() {}
    Object(sf::Vector2f position, sf::Vector2f velocity,
     float angle, sf::Vector2f size, float scale) :
     position(position), velocity(velocity), angle(angle),
     size(size), scale(scale) { }

    virtual void draw(sf::RenderWindow& window) = 0;
    virtual void update() = 0;

    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f size;
    float angle = 0;
    float scale;
    bool Collided = false;

    sf::FloatRect getBoundingBox();
    bool isOOB();
};
```

Appendix 18

Contents of the file 'Object.cpp'

```
#include "Object.h"

bool Object::isOOB() {

    return (position.x > Params::WindowWidth || position.y >
     Params::WindowHeight || position.x < 0 || position.y <
     0);
}

sf::FloatRect Object::getBoundingBox() {
    return (sf::FloatRect(position, size * scale));
}
```

Appendix 19

Contents of the file 'Params.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include <string>
class Params {
public:
```

```
static float pi;

const static sf::Vector2f nullvec;

static int WindowWidth;
static int WindowHeight;
static int Framerate;

static bool GraphicalSimulation;

static int MaxSimulationTime;

static float PhysicsTimeStepsPerSecond;
static float PhysicsTimeStep;

// Rocket physics

static float EnginePower;
static float RotationalEnginePower;

static float ConstAirResistance;
static float Friction;

static float NNC_Deadzone;

static std::string BackgroundTexture;

static std::string BlueNTT;
static std::string BlueFTT;
static std::string RedNTT;
static std::string RedFTT;

static std::string TargetT;


static sf::Vector2f posRocketNN;
static sf::Vector2f posRocketOPP;
static sf::Vector2f posRocketHC;
static sf::Vector2f posRocketDMM;

static float scaleRocket;
static float scaleTarget;

static sf::Vector2f sizeTarget;

static sf::Vector2f posTarget;
static float radTarget;

static float angleRocketNN;
static float angleRocketOPP;
static float angleRocketHC;
static float angleRocketDMM;
static float angleTarget;

static int nr0fInputs;
static int nr0fOutputs;
static int nr0fHiddenLayers;
static int NeuronsPerHiddenLayer;
static int BiasValue;

//GenAlg
static int MaxGenerations;
static int WeightCount;
```

```
    static int PopulationSize;

    static float FitnessResolution;

    static std::string SaveLocation;

};
```

Appendix 20

Contents of the file 'Params.cpp'

```
#include "Params.h"

float Params::pi = 3.1415926535;

const sf::Vector2f Params::nullvec = sf::Vector2f(0, 0);

// Render properties
int Params::WindowWidth = 1200;
int Params::WindowHeight = 600;
int Params::Framerate = 60;

bool Params::GraphicalSimulation = true;

int Params::MaxSimulationTime = 5;

//Qtsma
float Params::PhysicsTimeStepsPerSecond = 60;
float Params::PhysicsTimeStep = 1.0f /
 PhysicsTimeStepsPerSecond;

// Rocket physics
float Params::EnginePower = 1.0f;
float Params::RotationalEnginePower = 0.015f;

float Params::ConstAirResistance = 0.003;
float Params::Friction = 0.8;

float Params::NNC_Deadzone = 0.05;    // This value determines
the deadzone for Left-Right controls of the Neural Net.

std::string Params::BackgroundTexture =
 "../images/background.png";

std::string Params::BlueNTT =
 "../images/resized_by5/BlueNT.png";
std::string Params::BlueFTT =
 "../images/resized_by5/BlueFT.png";
std::string Params::RedNTT =
 "../images/resized_by5/RedNT.png";
std::string Params::RedFTT =
 "../images/resized_by5/RedFT.png";

std::string Params::TargetT = "../images/TargetT.png";

// Initial positions of the rockets
sf::Vector2f Params::posRocketNN =
 sf::Vector2f(Params::WindowWidth / 2, Params::WindowHeight
 / 2);
sf::Vector2f Params::posRocketOPP =
 sf::Vector2f(Params::WindowWidth - 150,
 Params::WindowHeight - 100);
```

```
sf::Vector2f Params::posRocketHC =
 sf::Vector2f(Params::WindowWidth / 2, Params::WindowHeight
 * 3 / 4);
sf::Vector2f Params::posRocketDMM =
 sf::Vector2f(Params::WindowWidth - 150,
 Params::WindowHeight - 100);

float Params::scaleRocket = 0.2;
float Params::scaleTarget = 0.132;

sf::Vector2f Params::sizeTarget = sf::Vector2f(150.0f,
 150.0f);

sf::Vector2f Params::posTarget =
 sf::Vector2f(Params::WindowWidth / 10, Params::WindowHeight
 / 2);
float Params::radTarget = 75;

float Params::angleRocketNN = 1.75 * pi;
float Params::angleRocketOPP = pi;
float Params::angleRocketHC = 1.5 * pi;
float Params::angleRocketDMM = 0;
float Params::angleTarget = 0;

/* Neural Net Specifications */
int Params::nr0fInputs = 5;
int Params::nr0fOutputs = 2;
int Params::nr0fHiddenLayers = 1;
int Params::NeuronsPerHiddenLayer = 6;

int Params::WeightCount = nr0fInputs * NeuronsPerHiddenLayer
 + (pow(NeuronsPerHiddenLayer, (nr0fHiddenLayers - 1))) *
 (nr0fHiddenLayers -1) + nr0fOutputs * NeuronsPerHiddenLayer
 + (NeuronsPerHiddenLayer * nr0fHiddenLayers + nr0fOutputs);

int Params::MaxGenerations = 1000000;

int Params::BiasValue = 1.0f;

int Params::PopulationSize = 100;

float Params::FitnessResolution = 4;

std::string Params::SaveLocation = "./Save/";
```

Appendix 21

Contents of the file 'PerformanceRun.h'

```
#pragma once
#include "Manager.h"

class PerformanceRun : public Manager {
public:
    PerformanceRun() {}

    virtual bool Simulate(float angle) override;

};
```

Appendix 22

Contents of the file 'PerformanceRun.cpp'

```
#include "PerformanceRun.h"

bool PerformanceRun::Simulate(float angle) {

    I1.angle = angle;

    int NrOfUpdates = 0;

    NN1.feedWeights(POP1.Genomes[iGenome].value);

    while (true) {
        if (I1.Collided || B1.Collided) { break; }

        if (I1.isOOB() && B1.isOOB()) { break; }

        if (NrOfUpdates / Params::PhysicsTimeStepsPerSecond
         >= Params::MaxSimulationTime) { break; }

        I1.SetNNControls(&NN1);
        I1.update();

        B1.update();

        TargetB1.update();

        NrOfUpdates++;
    }

    float SimulationTime = NrOfUpdates /
     Params::PhysicsTimeStepsPerSecond;    // in seconds

    if (POP1.Genomes[iGenome].fitness == 0)
        POP1.Genomes[iGenome].fitness =
         I1.calcFitness(SimuTimeInSec);
    else
        POP1.Genomes[iGenome].fitness +=
         I1.calcFitness(SimuTimeInSec);

    return false;
}
```

Appendix 23

Contents of the file 'Population.h'

```
#pragma once
#include <vector>
#include "Utilities.h"
#include "Params.h"
#include <iostream>

struct sGenome {

    std::vector<float> value;
    float fitness = 0;

    sGenome() : fitness(0) { for (int i = 0; i <
     Params::WeightCount; ++i) { value.push_back(0.0f);    }
}

    sGenome(std::vector<float> inputvector) :
     value(inputvector) {}
    sf::Vector2f initposI1;
    sf::Vector2f initposB1;
    sf::Vector2f initposTargetB1;

};


class Population {
private:
    sGenome BuildRandomGenome();

    sGenome Roulette();
    void SortPopulation();
    std::vector<sGenome> Crossover2(std::vector<sGenome>);
    std::vector<sGenome> Mutate2(std::vector<sGenome>);
    std::vector<sGenome> pickBests(int topN, int copies);


public:

    Population() { BuildRandomPopulation(); }
    Population(std::vector<sGenome> inputVector) { Genomes =
     inputVector;    }

    std::vector<sGenome> Genomes;
    float AverageFitness;

    void BuildRandomPopulation();

    void Evolve();

    float CalculateAverageFitness();
    float getBestFitness();

};
```

Appendix 24

Contents of the file 'Population.cpp'

```
#include <vector>
#include <iostream>

#include "Population.h"


// ************ Population ***************//
sGenome Population::BuildRandomGenome() {

    std::vector<float> RandomGenome;

    for (int i = 0; i < Params::WeightCount; ++i) {
        RandomGenome.push_back(randfloat(-1, 1));
    }

    return sGenome(RandomGenome);
}

void Population::BuildRandomPopulation() {

    for (int i = 0; i < Params::PopulationSize; ++i) {
        Genomes.push_back(BuildRandomGenome());
    }
}

void Population::SortPopulation() {

    std::sort(Genomes.begin(), Genomes.end(), [](const
     sGenome& lhs, const sGenome& rhs) {
        return lhs.fitness > rhs.fitness;
    }
    );
}

float Population::CalculateAverageFitness() {


    float sum = 0.0f;

    for (int i = 0; i < Params::PopulationSize; ++i) {
        sum += Genomes[i].fitness;
    }

    return sum / Params::PopulationSize;

}

float Population::getBestFitness() {

    return std::max_element(Genomes.begin(), Genomes.end(),
     [](const sGenome& lhs, const sGenome& rhs) {
        return lhs.fitness < rhs.fitness;
    }
    ) ->fitness;

}

std::vector<sGenome> Population::pickBests(int topN, int
 copies) {

    std::vector<sGenome> returnvector;

    for (unsigned i = 0; i < topN; ++i) {

        for (unsigned j = 0; j < copies; ++j) {

            returnvector.push_back(Genomes[i]);
        }
    }
    return returnvector;
}

void Population::Evolve() {

    SortPopulation();

    std::vector<sGenome> NewPopulation = pickBests(4, 1);

    while (NewPopulation.size() != Genomes.size()) {

        std::vector<sGenome> Parents;
        Parents.push_back(Roulette());
        Parents.push_back(Roulette());

        while (Parents[0].value == Parents[1].value)
```

```
                Parents[1] = Roulette();

            std::vector<sGenome> ParentsCrossedOver =
             Crossover2(Parents);
            std::vector<sGenome> SpecimensMutated =
             Mutate2(ParentsCrossedOver);

            NewPopulation.push_back(SpecimensMutated[0]);
            NewPopulation.push_back(SpecimensMutated[1]);
        }
        Genomes = NewPopulation;

        for (sGenome& genome : Genomes) {
            genome.fitness = 0;
        }
    }

    std::vector<sGenome>
    Population::Crossover2(std::vector<sGenome> Parents) {

        float CrossoverRate = 0.7f;

        if (randfloat() <= CrossoverRate) {
            int RandomCrossoverPoint = rand() %
             Params::WeightCount;

            for (int i = RandomCrossoverPoint; i <
             Params::WeightCount; ++i) {
                std::swap(Parents[0].value[i],
                 Parents[1].value[i]);
            }

            std::vector<sGenome> NewParents;
            NewParents.push_back(Parents[0]);
            NewParents.push_back(Parents[1]);

            return NewParents;
        }
        else
            return Parents;
    }

    std::vector<sGenome>
    Population::Mutate2(std::vector<sGenome> Specimens) {

        float MutationRate = 0.005;

        bool ChangesDone = false;

        for (int i = 0; i < Specimens[0].value.size(); ++i) {
            if (randfloat() <= MutationRate) {
                Specimens[0].value[i] = 1 -
                 Specimens[0].value[i];
                ChangesDone = true;
            }
        }
        for (int i = 0; i < Specimens[1].value.size(); ++i) {
            if (randfloat() <= MutationRate) {
                Specimens[1].value[i] = 1 -
                 Specimens[1].value[i];
                ChangesDone = true;
            }
        }
        if (ChangesDone) {
            std::vector<sGenome> returnvector;
            returnvector.push_back(Specimens[0]);
            returnvector.push_back(Specimens[1]);

            return returnvector;
        }
            return Specimens;
    }

    sGenome Population::Roulette() {

        float Sum = 0.0f;
        float FitnessSoFar = 0.0f;

        for (int i = 0; i < Params::PopulationSize; ++i) {
            Sum += Genomes[i].fitness;
        }

        double Slice = (double)(randfloat() * Sum);

        for (int i = 0; i < Params::PopulationSize; ++i) {
            FitnessSoFar += Genomes[i].fitness;

            if (FitnessSoFar >= Slice)
                return Genomes[i];
        }
        return Genomes.back();

    }
```

Appendix 25

Contents of the file 'RocketController.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Object.h"
#include "Utilities.h"
#include "RocketHUD.h"
#include <string>


class RocketController : public Object {
public:
    RocketController() {}
    RocketController(sf::Vector2f position, float angle,
     std::string ntTexture, std::string ftTexture) :
     Object(position, Params::nullvec, angle,
     sf::Vector2f(45, 45), Params::scaleRocket),
     noThrottleString(ntTexture),
     fullThrottleString(ftTexture) {

        noThrottle.loadFromFile(noThrottleString);
        noThrottle.setSmooth(true);
        noThrottle.setRepeated(false);

        fullThrottle.loadFromFile(fullThrottleString);
        fullThrottle.setSmooth(true);
        fullThrottle.setRepeated(false);
    }
    virtual void controls() = 0;

    virtual void update() override;
    virtual void draw(sf::RenderWindow& window) override;

    float throttle = 0.0f;
    void accelerate(float amount);

    void DefineTarget(Object * EnemyRocket);
    Object * LockOnTarget;

    float angular_throttle = 0.0f;
    void angular_accelerate(float amount);
    float angular_velocity = 0.0f;

    void CheckForSpin(
    bool SpinAlert = false;

    void TargetHit();

    float prevAngle = angle;
    float rotationalSum = 0.0f;

    sf::Texture noThrottle;
    std::string noThrottleString;
    sf::Texture fullThrottle;
    std::string fullThrottleString;
};
```

Appendix 26

Contents of the file 'RocketController.cpp'

```
#include "RocketController.h"
#include "Params.h"
#include <iostream>    // don't forget to remove

void RocketController::draw(sf::RenderWindow& window) {

    sf::Texture* rocketTexture;

    if (throttle == 0.0f) {
        rocketTexture = &noThrottle;
    }
    else {
        rocketTexture = &fullThrottle;
    }

    sf::Sprite rocketSprite;
    rocketSprite.setTexture(*rocketTexture);
    rocketSprite.setPosition(position);
    rocketSprite.setRotation((angle + Params::pi / 2) *
     (180.0f / Params::pi));
    rocketSprite.setOrigin(19, 27);

    rocketSprite.setScale(Params::scaleRocket,
     Params::scaleRocket);

    window.draw(rocketSprite);
    RocketHUD HUD(window, *this);
}
```

```
void RocketController::update() {

    float angular_acceleration = angular_throttle *
     Params::RotationalEnginePower;

    controls();        // event poll here

    angular_velocity += angular_acceleration;
    angular_velocity = clamp(angular_velocity, -0.4f, 0.4f);
    angular_velocity *= Params::Friction;

    prevAngle = angle;
    angle += angular_velocity;

    rotationalSum += angle - prevAngle;

    if (angle < 0)
        angle = 2 * Params::pi + angle;

    angle = std::fmod(angle, 2 * Params::pi);

    sf::Vector2f acceleration(cos(angle), sin(angle));
    acceleration *= throttle * Params::EnginePower;
    velocity += acceleration;
    velocity *= Params::Friction;
    position += velocity;
}

void RocketController::TargetHit() {

        Collided = getBoundingBox().intersects(LockOnTarget-
         >getBoundingBox());

}
void RocketController::accelerate(float amount) {

    throttle += amount;
    throttle = clamp(throttle, 0.0f, 1.0f);
}

void RocketController::angular_accelerate(float
 alpha_amount) {

    if (angular_throttle > 0 && alpha_amount < 0) {
     alpha_amount *= 3;
    }
    if (angular_throttle < 0 && alpha_amount > 0) {
     alpha_amount *= 3;
    }

    angular_throttle += alpha_amount;
    angular_throttle = clamp(angular_throttle, -1.0, 1.0);
}

void RocketController::DefineTarget(Object * EnemyRocket) {
    LockOnTarget = EnemyRocket;
}

void RocketController::CheckForSpin() {
    if (SpinAlert == false) {
        if (rotationalSum >= Params::pi) SpinAlert = true;
        else if (rotationalSum <= -1 * Params::pi) SpinAlert
         = true;
        }
    }
```

Appendix 27

Contents of the file 'RocketHC.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "RocketController.h"
#include <string>

class RocketHC : public RocketController {
public:
    RocketHC() {}
    RocketHC(sf::Vector2f position, float angle) :
     RocketController (position, angle, Params::BlueNTT,
     Params::BlueFTT) {}

    virtual void controls() override;

};
```

Appendix 28

Contents of the file 'RocketHC.cpp'

```
#include <iostream>
#include <SFML/Graphics.hpp>

#include "RocketHC.h"
#include "Utilities.h"
#include "Params.h"

void RocketHC::controls() {

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up)) {
        accelerate(0.01f);
    }
    else accelerate(-0.006f);

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
        if (angular_throttle > 0)
            angular_accelerate(-0.03f);
        else
            angular_accelerate(-0.01f);
    }
    else if
     (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
        if (angular_throttle < 0)
            angular_accelerate(0.03f);
        else
            angular_accelerate(0.01f);
     }
    else {
        angular_accelerate((-1 * angular_throttle) / 30);
        //angular_throttle = 0;
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down)) {
        accelerate(-0.33f);
    }
}
```

Appendix 29

Contents of the file 'RocketNN.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "RocketController.h"
#include "Params.h"
#include "NeuralNet.h"

class RocketNN : public RocketController {
public:
    RocketNN() {}

    RocketNN(sf::Vector2f position, float angle) :
     RocketController(position, angle, Params::BlueNTT,
     Params::BlueFTT) {}


    virtual void controls() override;
    virtual void update() override;

    virtual std::vector<float> getNNinputs() = 0;

    void calcDistance();
    float ClosestDistanceToTarget =
     sqrt(pow(Params::WindowHeight, 2) +
     pow(Params::WindowWidth, 2));

    float calcLookAtScore();
    float LookAtScore = 0;
    float normalizedLookAt();

    std::vector<float> NNInputs;

    float calcFitness(float SimulationTime);

    std::vector<float> NNControls;

    void SetNNControls(NeuralNet *NN);

};
```

Appendix 30

Contents of the file 'RocketNN.cpp'

```
#include "RocketNN.h"
#include "Utilities.h"
#include <SFML/Graphics.hpp>

void RocketNN::update() {
    RocketController::update();

    calcDistance();
    calcLookAtScore();
    CheckForSpin();
    TargetHit();
}

void RocketNN::controls() {

    if (NNControls[0] > 0.4)
```

```
            accelerate(0.01f);
        else
            accelerate(-0.006f);

        if (NNControls[1] < 0.5 - Params::NNC_Deadzone)
            angular_accelerate(-0.01f);
        else if (NNControls[1] > 0.5 + Params::NNC_Deadzone)
            angular_accelerate(0.01f);
        else
            angular_accelerate((-1 * angular_throttle) / 30);
    }

    void RocketNN::SetNNControls(NeuralNet *NN) {
        NNControls = NN->evaluate(getNNinputs());
    }



    void RocketNN::calcDistance() {

        ClosestDistanceToTarget = (sqrt(pow((LockOnTarget->
         position.x - position.x), 2) + pow((LockOnTarget->
         position.y - position.y), 2)));
    }

    float RocketNN::normalizedLookAt() {

        float Egocent_x = LockOnTarget->position.x - position.x;
        float Egocent_y = LockOnTarget->position.y - position.y;

        float Angle_reltoX = atan2(Egocent_y, Egocent_x);
        float Difference = 2 * Params::pi - angle - Angle_reltoX;

        float wrappedDifference = wrapRange(Difference, -
         Params::pi, Params::pi);

        return normalize(wrappedDifference, -Params::pi,
         Params::pi);

    }

    float RocketNN::calcLookAtScore() {

        float LookAtEnemy = normalizedLookAt();

        if (LookAtEnemy > 0.48 && LookAtEnemy < 0.52)
            LookAtScore++;
        else
            LookAtScore--;

        return 0.0f;
    }

    float RocketNN::calcFitness(float SimulationTime) {

        if (Collided)
            return normalize(Params::MaxSimulationTime –
             SimulationTime, 0, Params::MaxSimulationTime) + 2;
        else
            return normalize(1400 - ClosestDistanceToTarget, 0,
             1400);

    }
```

Appendix 31

Contents of the file 'Sandbox.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Params.h"
#include "RocketHC.h"

class Sandbox {
public:
    Sandbox() : Window(sf::VideoMode(Params::WindowWidth,
     Params::WindowHeight), "Sandbox Mode", sf::Style::None)
    {
        UserRocket = RocketHC(Params::posRocketHC,
        Params::angleRocketHC);

    }

    sf::RenderWindow Window;

    void HandleUserInput();

    void Run();

    float fps;
    float fpsLimit;

    RocketHC UserRocket;
    //RocketRND Bandit;

    bool pressedKeys[sf::Keyboard::KeyCount] = { false };
};
```

Appendix 32

Contents of the file 'Sandbox.cpp'

```
#include "Sandbox.h"
#include "GUI.h"
#include "Utilities.h"

void Sandbox::Run() {
    sf::Clock Clock;

    //bool CollisionDetection = false;
    //bool TargetHit = false;

    float PhysicsTimeStepAccumulator = 0.0f;

    while (Window.isOpen()) {

        const sf::Time FrameTime = Clock.restart();
        float FrameSeconds = FrameTime.asSeconds();
        fps = 1 / FrameSeconds;
        if (FrameSeconds > 0.1f) FrameSeconds = 0.1f;
        PhysicsTimeStepAccumulator += FrameSeconds;

        HandleUserInput();

        while (PhysicsTimeStepAccumulator >=
         Params::PhysicsTimeStep) {
            UserRocket.update();

            PhysicsTimeStepAccumulator -=
             Params::PhysicsTimeStep;
        }

        Window.clear(sf::Color::Black);
        UserRocket.draw(Window);
        Window.display();
    }
}

void Sandbox::HandleUserInput() {

    sf::Event event;

    while (Window.pollEvent(event)) {

        switch (event.type) {

        case sf::Event::Closed:
            Window.close();
            break;
        case sf::Event::KeyPressed:

            pressedKeys[event.key.code] = true;

            switch (event.key.code) {
            case sf::Keyboard::W:

                break;
            case sf::Keyboard::S:

                break;
            case sf::Keyboard::A:

                break;
            case sf::Keyboard::D:

                break;
            case sf::Keyboard::R:
                //UserRocket.reset();
                break;
            case sf::Keyboard::Add:
                if (Params::PhysicsTimeStep >= 0.003)
                    Params::PhysicsTimeStep -= (0.1 /
                     Params::PhysicsTimeStepsPerSecond);
                break;
            case sf::Keyboard::Subtract:
                Params::PhysicsTimeStep += (0.1 /
                 Params::PhysicsTimeStepsPerSecond);
                break;
            case sf::Keyboard::Escape:
                Window.close();
                break;
            }
            break;
        }
    }
}
```

Contents of the file 'Target.h'

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Params.h"
#include "Object.h"

class Target : public Object {
public:
    Target() : Target(Params::posTarget, Params::radTarget,
     Params::sizeTarget, Params::scaleTarget) {}

    Target(sf::Vector2f position, float radius, sf::Vector2f
     size, float scale) : Object(position, Params::nullvec,
     Params::angleTarget, size, scale), radius(radius),
     TextureString(Params::TargetT) {

        BodyTexture.loadFromFile(TextureString);
        BodyTexture.setSmooth(true);
        BodyTexture.setRepeated(false);

    }

    virtual void draw(sf::RenderWindow&) override;
    virtual void update() override;

    std::string TextureString;
    sf::Texture BodyTexture;
    sf::Sprite BodySprite;
    sf::CircleShape Body;
    float radius;
    void reset();

    bool randomized = true;
};
```

Appendix 34

Contents of the file 'Target.cpp'

```
#include "Target.h"
#include <iostream>


void Target::reset() {

    int randX = rand() % (int(Params::WindowWidth / 10)) +
     (int(Params::WindowWidth * 0.05));
    int randY = rand() % (int(Params::WindowHeight * 0.8)) +
     (int(Params::WindowHeight * 0.1));

    position = sf::Vector2f(randX, randY);
    Collided = false;
}

void Target::draw(sf::RenderWindow& Window) {

    sf::Texture* TargetTpointer;
    TargetTpointer = &BodyTexture;

    BodySprite.setTexture(*TargetTpointer);
    BodySprite.setPosition(position);
    BodySprite.setOrigin(Params::radTarget,
Params::radTarget);

    BodySprite.setScale(Params::scaleTarget,
Params::scaleTarget);

    Window.draw(BodySprite);
}

void Target::update() {
    BodySprite.setPosition(position);
    BodySprite.setOrigin(Params::radTarget / 2,
     Params::radTarget / 2);
}
```

Appendix 35

Contents of the file 'Utilities.h'

```
#pragma once
#include <cstdlib>
#include <SFML/Graphics.hpp>

float Sigmoid(float activation, float response = 1.0f);

float clamp(float x, float min, float max);
float rad2deg(float);
float deg2rad(float);
float normalize(float x, float min, float max);
float wrapRange(float x, float min, float max);
float CalculateDistance2(sf::Vector2f a, sf::Vector2f b);
std::vector<std::string> explode(std::string const &
 FullString, char Separator);

struct SPoint {
    float x, y;

    SPoint() {}
    SPoint(float a, float b) : x(a), y(b) {}
};

inline int randInt(int x, int y) {
    return rand() % (y - x + 1) + x;
}

inline float randfloat() {
    return (rand()) / (RAND_MAX + 1.0);
}

inline float randfloat(float x, float y) {
    float RandBase = ((float)rand() / (float)RAND_MAX);
    float range = y - x;
    float random = RandBase * range;
    return x + random;
}

inline bool randBool() {
    if (randInt(0, 1)) return true;
    else return false;
}

inline float randClamped()    { // -1 < n < 1
    return randfloat() - randfloat();
}
```

Appendix 36

Contents of the file 'Utilities.cpp'

```
#include "Utilities.h"
#include <sstream>

float clamp(float x, float min, float max) {

    if (x > max) {
        return max;
    }
    if (x < min) {
        return min;
    }
    else {
        return x;
    }
}

float rad2deg(float par) {
    return (par * 180 / 3.141592);
}

float deg2rad(float par) {
    return (par * 3.141592 / 180);
}

float normalize(float x, float min, float max) {
    return ((x - min) / (max - min));
}

float wrapRange(float x, float min, float max) {
    const double width = max - min;
    const double offsetValue = x - min;
    return (offsetValue - (floor(offsetValue / width) *
     width)) + min;
}

float CalculateDistance2(sf::Vector2f a, sf::Vector2f b) {
    return (sqrt(pow((b.x - a.x), 2) + pow((b.y - a.y), 2)));
}


std::vector<std::string> explode(std::string const &
 FullString, char Separator) {

    std::vector<std::string> result;
    std::istringstream iss(FullString);

    for (std::string token; std::getline(iss, token,
     Separator); ) {
        result.push_back(std::move(token));
    }

    return result;
}

float Sigmoid(float activation, float response) {

    activation *= response;
```

```
            float fastSigmoid = (activation / (1 + abs(activation)));
            return (fastSigmoid + 1) / 2;
        }
```