Alan Byju, Paul Geoghegan, Saul Burgess

**Oberon V2.05 Design Document**



Contents

[Overview 2](#_Toc33462474)

[ Introduction 2](#_Toc33462475)

[ Our inspiration 2](#_Toc33462476)

[ Design 2](#_Toc33462477)

[ Problems and Issues 2](#_Toc33462478)

[Attacking 3](#_Toc33462479)

[ Targeting 3](#_Toc33462480)

[ Firing 3](#_Toc33462481)

[Movement 4](#_Toc33462482)

[ Co-ordinate Generation 4](#_Toc33462483)

[ Main Loop 5](#_Toc33462484)

[ Go To function 5](#_Toc33462485)

# Overview

## Introduction

In this design document we will disclose the different aspects of our robot Oberon. Java was used as the language to implement the robot. This document will include the various techniques and formulas the robot uses. This document will also break down all the steps and processes that were undertaken to complete the robot.

## Our inspiration

The name Oberon is derived from the king of fairies according to medieval and Renaissance literature. Most famously known for its appearance in the play *Midsummers Night,* by William Shakespeare. This name was chosen because the fairy king Oberon represents the forest and trees, which seemed appropriate since we used a decision tree as Oberon’s primary strategy. This relationship between the forest and fairies also inspired us to choose green as our robot's livery.

## Design

The design and coding of the robot required the analysis of the rules and physics of Robocode, to do this we created a flowchart which took into consideration every event and situation that the robot would encounter. The flowchart helped us to figure each of the characteristics of our robot, like movement, targeting and dodging.

Based on analysis of the flowchart there were several changes that needed to be resolved. These changes were implemented straight into the code. An example of such an improvement was the random movement, In the beginning Oberon would venture into the Border Zone, leading to its destruction via the Sentry Robot. We will expand upon how we solved this later in Movement.

## Problems and Issues

From the very beginning of the design process, we faced many trials. Beginning with misreading the rules: until a few weeks before the TUD (Technological University Dublin) competition Oberon was an advanced robot. This problem would have resulted in not being able to compete in the competitions. Once we discovered this issue, we worked to refactor the robot from an advanced robot to a classic robot.

Due to the above problem, we had several issues with reaction time and general sluggishness. Over time we optimized the code, reducing these problems until we were confident in the efficiency of our robot.

# Attacking

## Targeting

Oberon fires using a simple 360-degree radar turn. When *onScannedRobot* is triggered the firing solution algorithm is triggered, causing the scanner to stop until the firing solution calculation is triggered. This leads to a ‘Gatling gun’ effect when the enemy tank remains still. We feel this is a key component to our combat power.

**360 Scan -> Stop on See Robot -> Trigger Firing Calc -> Fire**

## Firing

Oberon uses a ‘Linear Targeting’ Algorithm as its method of calculating a firing solution. This system projects forward the position the enemy will be, and fires at this projected position. We based this on the speed of a 3.0 power bullet.

Linear targeting relies on the use of trigonometric functions, this is possible as we assume that there are three points between us and the enemy.

* Us, which is calculated by the *getX* and *getY* functions.
* Enemy, which is calculated using *ScannedRobotEvent*.
* Projected Position, which is calculated by multiplying lateral velocity by bullet travel time.

(hypotenuse, I.E distance between us and the projected position, is calculated using bullet speed multiplied by bullet travel time)

Using these three points, and liberal use of *normalRelativeAngle* this allows us to calculate the optimum angle to fire at to intersect. Note the calculation present in our code is a *Maximum Escape Angle*variant of Linear targeting, this means that in addition to firing at the projected point, it fires at the largest offset angle that could possibly hit the enemy robot, provided the physics of Robocode.

# Movement

## Coordinate Generation

Oberon’s movement is based on a system of randomly generated co-ordinates which are passed to a movement function which tells Oberon where to go.

At the start of each battle Oberon will generate an array which will dictate the co-ordinates that Oberon goes to.

This uses the formula:

**double range = getBattleFieldWidth() - (getSentryBorderSize() \* 2)**

This declares and initialized a double called *rangeX*, in turn this contains the value:

The width of the battlefield minus (the size of the sentry border times 2), this assumes that the battlefield is symmetrical.

This defines the maximum area that co-ordinates can be generated in. In an arena which is 800 X 800 with a sentry border of 300 this would leave Oberon with a play area of 200 X 200. After this Oberon divides up the playable area by in this case 5 which will be multiplied by itself to equal 25 as we found having more co-ordinates than 25 can slow it down.

Then a for loop executes which will set the co-ordinates that Oberon can generate in the goTo function.

This looks like:

A description of code:


if (getVelocity() < 1){

x= goToX[rand.nextInt(5)] - getX();

y = goToY[rand.nextInt(5)] - getY();

This loop executes 5 times to co-inside with the amount we divided the original block by. GoToX[i] sets the place in which this co-ordinate will be generated,

Range is divided by 4 so that Oberon will always stay at least the value of range divided by 4, Then range times I is added to this value. And is finally added to the sentry border size so that this co-ordinate is outside of the sentry’s zone of control.

After this loop has finished Oberon has generated a total of 25 co-ordinates.

## Main Loop

Oberon’s main loop will check if Oberon has stopped moving or is close to stopping by checking if its velocity is below 1. After it confirms this it will generate 2 random numbers between 0 and for which it will use inside of the square brackets of our go TOX and goToY arrays to pick 2 co-ordinates, one on the X axes and one on the Y axes.

It then stores these values in x and y which are then passed to the goTo function.

## GoTo function

Combined with the Coordinate generation system, Oberon moves between generated points on the map. It achieves this through a similar method to Linear Targeting, by generating a vector between (us) and (target point). It achieves this by first converting our position into two vector values (x, y) and the angle to the target position by using the Java atan function. After this is done, the goTo function calculates the turn between us and the target point, using the previous two datapoints. Once this is done, it is simply passed to the turning command and Oberon will turn in the desired direction. Moving to the point is done by simply getting the distance between the two points, then entering this into an ahead() command.