Boondoggle Developers Guide.

# Before You Begin.

Boondoggle is a programmer’s game where “drones” are built to compete against each other. To play boondoggle you build yourself a drone using code to write its control system. The drone is then sent into arenas where it will interact with other drones and the environment.

The programming of the drone determines how well it handles itself and how it deals with other bots and obstacles that it meets.

## Skills Required

You will need to know a little bit about programming, using C#.  
You will need to know how to clone a git repository.  
You will need to know how to compile a DLL from a C# Program

## Preparation

You will need a c# compiler and editor. The simplest is visual studio – as we have prepared a beginners solution there but any c# compiler and editor can be used.

Clone the getting started repository from here:

<https://github.com/Itsey/Boondoggle.Start.git>

When you clone the repository it will look like this:

|  |  |
| --- | --- |
|  | The docs folder contains this file and links to online documentation.  The DroneRink folder contains an executable that can be used to test your drones.  The src folder contains a reference implementation and example code for you to use to get started. |

Navigate to the src folder from the cloned repository and open the solution in Visual Studio.

|  |  |
| --- | --- |
| Table  Description automatically generated with low confidence | The \_Reference project contains this document and links to help as well as some copy and paste code.  The DroneExamples project contains examples of existing drones.  The MyAmazingDrone project is there for you to build your first drone. It starts with an empty file for the walkthrough. |

The solution will be able to be opened in visual studio. The solution contains two projects a \_Reference project which contains documentation and libraries and supporting tools and a c# assembly called AmazingBot. This assembly contains the sample code for three bots.

The amazing bot is an empty bot implementation in amazingbot.cs.

The demo bot is the code that is used in this document to create a functional bot.

The reference bot is sample code for a whole variety of features available in boondoggle.

Finally there is a supporting class – BotUtils.cs which contains helper methods to make working with the bots simpler. The helpers are used by the reference and DemoBots but they are entirely optional.

Open the solution and look at the code for DemoBot. We’ll walk through creating Demo Bot now.

# Walkthrough – Creating Your First Drone.

## Getting Started

The first thing you should do is compile the entire solution. From the Build Menu in Visual Studio choose rebuild solution. The solution should build cleanly.

Text, letter

Description automatically generated

Assuming that it builds clearly then follow the walk through to create your first drone.

## Step 1 – Your First Drone.

Rename MyAmazingDrone.cs to YourBotName.cs. Rename the class AmazingBot inside the file to your bot name but leave all the other methods the same for now.

|  |  |
| --- | --- |
| Graphical user interface, text, application, email  Description automatically generated | When you first open the file you will see that the most basic Drone possible is there, containing a single method and a name and a version.  Update line 20 in the file and change the name to any name you choose. It must be unique to you so try and think of something original.  For now we can leave everything else the same. |

While this is the minimum needed for a drone it won’t do anything as it has no power, or any logic to it at all. However we can still try and use it.

## Step 2 – Running The Drone Simulator.

Prior to sending your drone out into the wide world it is a good idea to give it a bit of a test, to see whether or not it is likely to do what you want. The Boondoggle world is large and complex but the simulator is small and tightly controlled so you can give your drone a good workout.

From the \_reference project you have a copy of the simulator ready to go.

|  |  |
| --- | --- |
| Graphical user interface, text, application  Description automatically generated | Locate the DroneTestCenter in your solution and open it in explorer.  Double click the BdDebugRenderer.Exe to launch the simulator. |

|  |  |
| --- | --- |
| Chart, treemap chart  Description automatically generated | To run the simulator you must enter the full path to your newly created drone into the highlighted box.  Once you do that and press “load” you will see your drone available for simulation.  Click your drone name and press start. |

|  |  |
| --- | --- |
| Graphical user interface, chart  Description automatically generated | When you press start you will see the world load and notice in the bottom left there is a small red dot. That is your drone!  Right now its just going to sit there doing absolutely nothing.  As you watch the turns will keep ticking by, you can change the battle speed and they will whiz by faster but still your drone does nothing. |

While we have a working drone its not exactly doing much. Lets fix that.

## Step 3 – Make Your Drone Do Things

Lets make one very simple change, if our drone is travelling at less than 5 then lets make it accelerate so that its at least moving around. Change the code to this (also in step3.txt in the Steps folder)

|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |

Once you have done that you can reload the drone into the control centre.

|  |  |
| --- | --- |
| Graphical user interface, application  Description automatically generated | Currently you have to close the centre and start it from scratch.  Press start when your drone is loaded.  Your drone should start moving. What happens next? |

Well ouch, what happened? Your drone drove into the wall, which reduced its speed to zero. It then thought it was going slowly and accelerated straight into the wall again. In fact it kept doing this until it died! Oops!

## Step 4 – Scanning And Navigating.

Obviously flying repeatedly into a wall isn’t going to cut it out there in the bad world of boondoggle, we are going to have to take it up a level. Your drone is equipped with a basic scanner and using the scanner as the “eyes” of the drone we can navigate around the map.

Lets add some code to detect if there is a wall ahead and if there is then change direction. Currently the logic simply states that the drone should accelerate to speed 5 and drive dead ahead. With a small change it will accelerate to speed 5 and avoid the impending walls:

|  |  |
| --- | --- |
|  | The new code uses a piece of equipment, retrieving the equipment by name.  Using a scanner returns a “ScanEquipmentUseResult” and this has some useful helpers to help you interrogate the scanner. |

Using the “IsWayAheadBlocked” method on the scan equipment use result means we can simply check for a wall up ahead. If we detect a wall up ahead we just turn 90 degrees to avoid it.

Go ahead and make the code changes and compile up the new bot and run it in the simulator. You’ll see it drive round and round the edge of the map until you get bored watching or the power runs out.

## Step 5 – The enemy!

Well we are happy, pootling round the edge of the arena. Happy but lonely, so lonely. Lets bring a friend along to play.

## Step 6 – Get Tooled Up.

In step 4 we looked at using a scanner, this was a piece of equipment installed in the bot which performed an action. There are multiple categories of equipment from powerpacks, scanners, weapons and utilities. Some of these are mandatory ( such as a powerpack ) and some are entirely optional such as weapons.

# Creating Demo Bot

## Getting Started

Rename AmazingBot.cs to YourBotName.cs. Rename the class AmazingBot inside the file to your bot name but leave all the other methods the same for now. The bot name can be any name you choose but it must be unique to you.

|  |  |
| --- | --- |
|  | When you first open the new Bot file you will see that there are three methods to override which is how you create your Bot.  See full API documentation. |

Currently if the Bot is compiled and run it will be a valid bot, but one which does not do anything. In order to make it do something we will first customise the DemoBot.

## Initialisation

While the initialisation for a Bot is optional, most Bots have equipment installed and announce their arrival into the game world with a custom message. Therefore the first thing that we do is to install some equipment in our Bot and customise the greeting message.

See this link for descriptions of equipment and types of equipment but for our first ever bot we will stick with the basics.

The most basic of equipment that a Bot requires is a PowerPack to provide energy to the Bot systems and a scanner to be aware of our surroundings. Finally it’s a dangerous world out there so it’s probably best if we equip a weapon.

Update ActualPrepareForBattle with the following code:

|  |  |
| --- | --- |
|  | The code here can be typed, copied from reference bot or copied and pasted from step1.txt in the guide folder |

The changes that we have made will set an introduction message (go on… change the string to anything you like) as well as install three basic pieces of equipment.

Each piece of equipment when it is installed has a type – specified by the enum as the first parameter, as well as a name which you can use to refer to the item in your code and finally a location. In the example code the scanner and power pack are mounted internally whereas the weapon is mounted in the turret. This is important as the turret is the only location with 360 degrees of fire – all of the other mount points have restrictions. See Weapons, Mountpoints and line of sight.

Our bot is now actually ready to enter the arena. Currently however it will just sit there and not do anything at all. We should probably make it at least try and do something.

# AI – or Make It Do Something

ActualTakeAction is where all of the hard work of the Bot goes, this is where the decisions that will affect the Bots behaviour are taken each tick and turn. Typical actions for a Bot include scanning its surroundings, changing speed or direction or opening fire on enemies.

There are no rules as to what you must or must not do in any given action – you have a certain amount of energy you can use but how you use it is up to you. In this example we will first take a scan of our surroundings to see what we are near.

## Scanning

When we installed the scanner in the code above we gave it a name – “myScanner” therefore we will now request to use a piece of equipment called “MyScanner”. If you changed the name in the sample code you’ll need to change it here too.

|  |
| --- |
|  |

Insert this line of code into ActualTakeAction, this will use the equipment that we named “MyScanner” and return the results as a “ScanEquipmentUseResult” into lastScan. As we have placed this in ActualTakeAction -it will occur every single tick, therefore we will scan 10 times per turn. This is very diligent but might not be desirable in the long run.

|  |  |
| --- | --- |
|  | To analyse how scan results are returned imagine the scenario here.  Your bot (the red square) uses a scanner that has a cone like forward facing view. The cone scans outward and forward from your position only.  In range is a solid wall running along the left hand side of your bot and there is another bot ahead and slightly to the right. The cone of the scan is highlighted in yellow.  The scan can see a portion of the wall and the other bot. |

When the scan takes place and the results are returned you receive an object which has a grid of scan results centred upon your bot. The grid will have the dimensions required in order to return the full scan result – in this case the cone stretches forward 4 squares and to the left and right four squares therefore the grid is 9 by 9, with your bot centred at the position 0,0 on the grid.

|  |  |
| --- | --- |
|  | The ScanResult.  GetResultAtPosition is called for each point you wish to know the contents of. The diagram here shows how the results would be returned.  All of the areas not scanned by this particular scanner return ScanResult.Unscanned – highlighted in grey.  Areas that are within the scan but contain nothing return ScanResult.Unoccupied. |

As you can tell, this particular scanner returns a lot of empty results relative to the number of tile that it has actually scanned. However different scanners have different properties and therefore will return different shapes and distances of scanning.

To determine if it is safe to accelerate we will check the results of the ScanResult, looking dead ahead – then if there is nothing there we will accelerate. To do this we will use a very simple helper method – IsThisDirectonClear. IsThisDirectionClear looks through a scan result for a given heading and determines if there are objects in the way. Lets add the code now:

## Finding Other Bots

Aside from avoiding obstacles and determining whether we can drive straight ahead the scan also shows things around the bot that are interesting, whether its loot to be salvaged, enemies to be engaged or other environmental features.

To save you hunting through all of the results of a scan, looking for interesting things the scan will return to you all points of interest. GetPointsOfInterest returns a list of points that contain something other than just an unscanned, unoccupied or solid wall space.

Lets add the code now to identify the other Bot in the scan and take a pot shot at it.

|  |  |
| --- | --- |
|  | Our bot code now looks for another bot and shoots at it if it is present. N.b. for simplicity the code to choose a speed is removed here but the full code is in step4.txt in the solution |

This simple example just shoots at the first bot it sees. The FireWeapon command requires the name of a weapon that we installed already and additionally the identity of the target. The target identity is returned by the Scan. N.B. this is a temporary identity which is only valid in the context of this single scan for this turn.

# Testing Your Bot

# What Next?

This is only the beginning, what your bot can do to outsmart, avoid or kill its opponents is up to you and your coding prowess.

# Advanced Topics

## The Game World

The Boondoggle world consists of a series of elements that interact to determine how the Bots behave. A Bot interacts with its environment and other Bots.

Turns and Ticks

Each unit of time in the game is a *turn*. A turn represents an in game second and is subdivided into ten *ticks*. A bot gets 10 chances to perform activities each turn and in addition a single chance prior to a new turn starting. Your Bots code must run in a timely fashion each tick, but gains a little more time during the turn preparation phase.

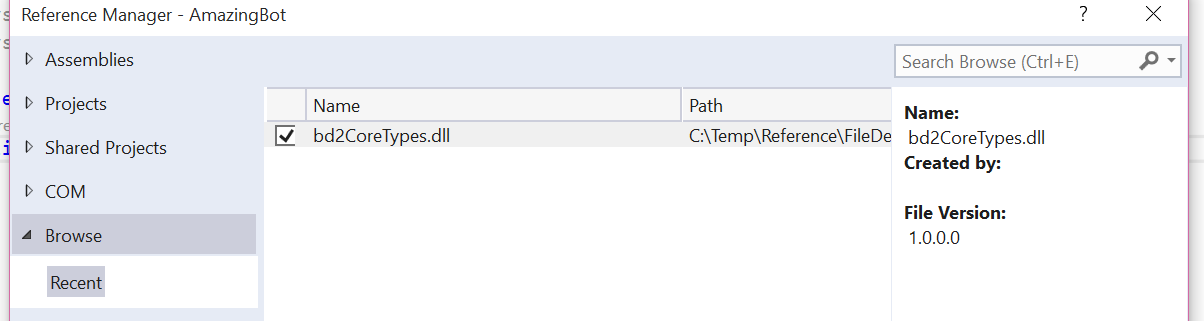
During each tick your bot may move, depending on its *speed* and *heading*. The speed at which a Bot is going determines things such as how far it moves, how hard it is to turn and how hard it is for other bots to hit. It will also determine the amount of damage received during a *collision*.

Appendix – Glossary

|  |  |
| --- | --- |
| Bot | A player’s avatar, a combination of an AI and an equipment loadout. |
| Turn | A unit of time in the game world, sub divided into tics |
| Tic | The smallest unit of time in the game world – there are 10 of these to each turn. |
| Collision | Occurs when two objects hit each other - either a Bot into a wall or a Bot into another Bot |
| Salvage | The loot that can be gained by scavenging components from destroyed Bots or other wreckage |
| PowerPac | The primary source of power for a Bot- every Bot requires a powerpac otherwise it can not move, or perform activities |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Appendix adding new class

Add reference to bdCoreTypes



Update namespace to be Boondoggle.Playah

Bot Builders Guide.

Game Structure

The game is divided into Turns, each turn represents a second of game world time and it is divided into 10 ticks. Your bot gets a chance to perform an activity each tick, and once just before each turn starts – therefore a total of 11 chances to do things each turn.

In addition the bot gets a single chance to operate prior to the battle beginning. This is a call to the BotPrepareForBattle method that you can override. Bots can perform significant operations in this call.

Bot Limits

|  |  |
| --- | --- |
| undefined | Bots are constrained by the size of the frame that they use, their power, weight and so on.  Each turn a bot has an amount of *charge* available to it – this is the amount of energy that a bot can consume in a single second (10 ticks) and it refils each turn.  Power on the other hand represents the total amount of power available to the bot and it does not refill.  When a bot runs out of power it is depleted, and therefore can no longer move or operate. A depleted bot is a dead bot so keep an eye on your total power and retreat before it’s all gone. |

Navigation.

Bots travel around their environment with a combination of *speed* and *heading*. Speed represents the rate at which a bot travels and heading is the direction using degrees where a heading of 0 is directly up the map.

|  |  |
| --- | --- |
|  | Heading  Heading can be represented as any number of degrees from 0 up to 359.  To set a new direction from within your bot call the base ChangeHeading() method passing in the desired heading.  N.b. the ChangeHeading method returns a value which is the new heading, its not always possible to turn at the rate that you wish. For example if you are travelling at high speed and try and turn through 180 degrees it is unlikely you will be able to do so in one change heading request. To check your new heading check the return of ChangeHeading or the CurrentHeading property. |

Navigating Round Corners

|  |
| --- |
| Beware when taking corners too tight, Boondoggle uses squares for its navigation therefore when you are driving past a curved wall at an angle it is possible that you will cut across the corner of the wall and collide even though you don’t think that should be the case.  In the example below the bot travels at 37 degrees but actually hits the wall as when you look closely the direction of travel intersects one of the wall pieces. |
|  |

Speed And Acceleration

Each bot has to have a powerpack equipped. This powerpack provides the source of all of the bots power including its acceleration and engine power. A power pack will determine how much a bot can accelerate by each turn as well as its top speed.

To request that the bot moves faster use the Accelerate base method. To reduce speed use the Decelerate base method. The default rate of change for these methods is 1 and just like the change heading method they return then actual speed that the bot ends up at.

The speed of a bot is how fast it travels within a single turn, therefore a bot may not move every tick. The slower that a bot travels the fewer times it moves each turn.

|  |  |
| --- | --- |
|  | The actual rate at which bots move at each speed is not documented but here is an example to illustrate how it works.  A bot travelling at speed 1 only moves on tick 5, whereas a bot travelling at speed 10 moves on every single tick. |