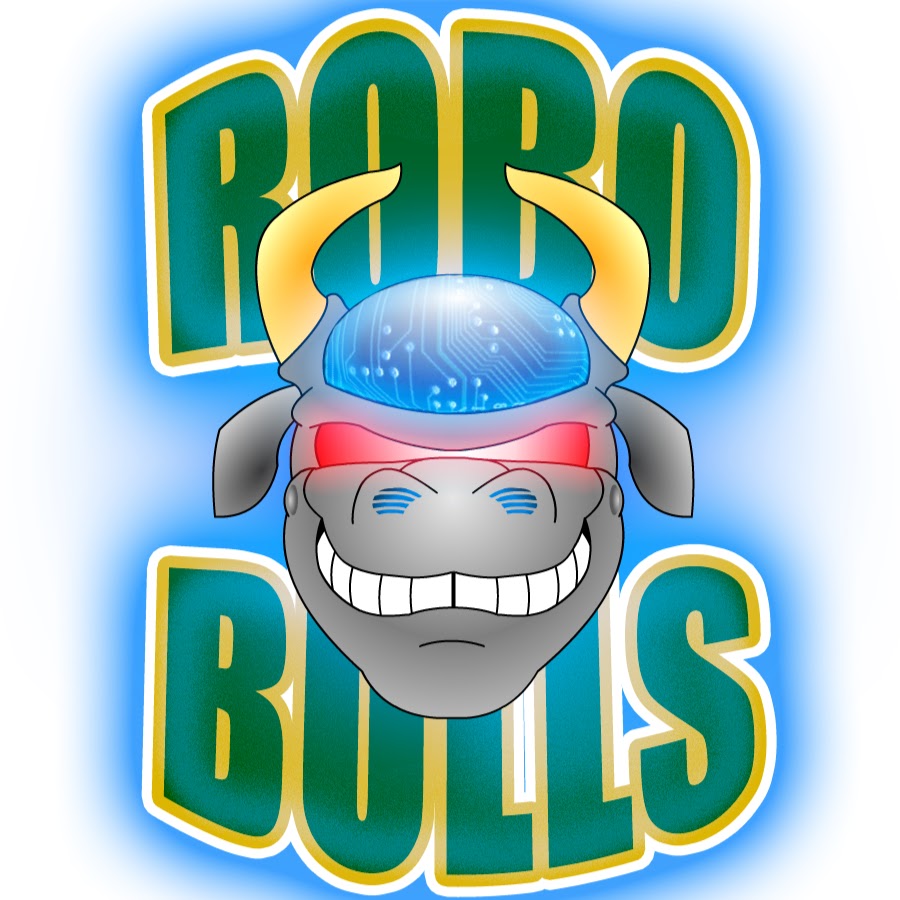
Robobulls

User Documentation

This document introduces what is the robobulls software and explains how to run it. [ The document was extended, TODO: modify this to explain provide a description of the contents of the document. In the meantime, see the index for an idea about the contents of this document. ]



# Table of Contents

[1 Table of Contents 2](#_Toc54203022)

[2 What is the robobulls software? 4](#_Toc54203023)

[3 Required software 4](#_Toc54203024)

[3.1 Grsim modification - accurate speed measurements 5](#_Toc54203025)

[4 Compiling the code. 6](#_Toc54203026)

[4.1 Windows 6](#_Toc54203027)

[4.2 Linux 8](#_Toc54203028)

[4.3 MAC 8](#_Toc54203029)

[5 Running robobulls software 9](#_Toc54203030)

[5.1 Starting the program 9](#_Toc54203031)

[5.2 How to set up robobulls configuration files. 10](#_Toc54203032)

[5.2.1 field.yaml 11](#_Toc54203033)

[5.2.2 motion.yaml 11](#_Toc54203034)

[5.2.3 comm.yaml 11](#_Toc54203035)

[5.2.4 team.yaml 12](#_Toc54203036)

[5.3 Connecting to SSL software and controlling the robots 13](#_Toc54203037)

[5.3.1 Connecting to SSL 13](#_Toc54203038)

[5.3.2 Controlling robots 14](#_Toc54203039)

[5.3.3 Playing a full game 14](#_Toc54203040)

[5.3.4 Debugging vision communication issues 14](#_Toc54203041)

[5.3.5 Debugging robot communication issues 14](#_Toc54203042)

[5.3.6 Debugging game controller communication issues 14](#_Toc54203043)

[6 GUI 15](#_Toc54203044)

[6.1 Gui components 15](#_Toc54203045)

[6.1.1 Panel Output 15](#_Toc54203046)

[6.1.2 Panel Game Info 15](#_Toc54203047)

[6.1.3 Panel field 16](#_Toc54203048)

[6.1.4 Panel Selected robot 16](#_Toc54203049)

[6.1.5 Panel Teams 17](#_Toc54203050)

[6.2 User interaction 17](#_Toc54203051)

[6.2.1 Dragging the field 17](#_Toc54203052)

[6.2.2 Robot selection 17](#_Toc54203053)

[6.2.3 Robot camera tracking 17](#_Toc54203054)

[6.2.4 Hot keys 17](#_Toc54203055)

[6.3 Other functions 18](#_Toc54203056)

[7 Robobulls debugging terminal 19](#_Toc54203057)

[8 Code 20](#_Toc54203058)

[8.1 Skills Required 20](#_Toc54203059)

[8.2 Main concepts 20](#_Toc54203060)

[8.3 Folder Structure 21](#_Toc54203061)

[8.4 Units of measurements 22](#_Toc54203062)

[8.5 Threads 22](#_Toc54203063)

[8.5.1 Debugging thread 23](#_Toc54203064)

[8.5.2 Game controller listener thread 23](#_Toc54203065)

[8.5.3 Vision listener thread 23](#_Toc54203066)

[8.5.4 GUI thread 23](#_Toc54203067)

[8.6 Class diagrams & class descriptions 24](#_Toc54203068)

[8.6.1 Game State 24](#_Toc54203069)

[8.6.2 Robot Control 26](#_Toc54203070)

[8.6.3 Team Control 28](#_Toc54203071)

[8.7 Adding code 29](#_Toc54203072)

[8.7.1 Adding support for new robots 29](#_Toc54203073)

[8.7.2 Adding strategy controllers 29](#_Toc54203074)

[8.7.3 Adding strategies 29](#_Toc54203075)

[8.8 SSL communication 29](#_Toc54203076)

[9 TODO List 30](#_Toc54203077)

[9.1 Tasks for new members 30](#_Toc54203078)

[9.2 Documentation 30](#_Toc54203079)

[9.3 Main code 30](#_Toc54203080)

[9.4 GUI 31](#_Toc54203081)

[9.5 grsim 31](#_Toc54203082)

# What is the robobulls software?

The robobulls software is designed to control up to 2 teams of robots playing soccer in the robocup[[1]](#footnote-1). The software was designed to be used in the SSL league but could easily be extended to work with other leagues as well. The following is a video from of a sample SSL league game.

[](https://www.youtube.com/embed/LpTRn8PD7GA?feature=oembed)

Figure 1 - Robocup game in the SSL league

# Required software

For the Robobulls software to work correctly, you will need other pieces of software provided by the SSL league[[2]](#footnote-2).

1. **SSL-game-controller**[[3]](#footnote-3) – used as a referee to control a game from a web browser. Binaries are available for any of the 3 major paltforms.
2. **SSL-vision**[[4]](#footnote-4) – used to detect the position of the ball and the robots on the field, and to transmit the information to the teams by means of multicast IP packages. This software is required only when using real robots and it is already installed in the biorobotics lab[[5]](#footnote-5). Do not install this software unless setting up a new field.
3. **grSim**[[6]](#footnote-6) – physics simulator used to simulate SSL games (only required for simulation). This software replaces SSL-vision in simulated environments, and provides TCP/IP interfaces to control their simulated robots. To use this software, you will need to download and compile their code. Instructions for linux and mac are provided on their website while we provide instructions for windows in the Robobulls documentation folder.
4. **SSL-vision-client**[[7]](#footnote-7) – Optional software that adds a graphical interface for visualizing vision packages to SSL-game-controller. Binaries are available for any of the 3 major platforms.

To successfully use Robobulls software, you will need to learn how to use each piece of software. Refer to their respective websites for detailed instructions on how to use them. Since you will most likely start by using a simulated environment, you can start by having a look at grSim and then learn the others as need arises.

To begin using Robobulls software, you will need at least one vision system (grSim if simulating or SSL-vision otherwise). You can control robots without the game controller, but eventually, to play a full game, you will need to use the software.

The SSL league updates the software regularly so you should monitor when changes are updated. Alternatively, you can subscribe to their notification system[[8]](#footnote-8) to receive the latest news.

In addition to the software provided by the SSL league, you will be required to get familiar with git[[9]](#footnote-9), a distributed versioning system. If you are not already familiar, as need arises, you should at least learn the basic commands[[10]](#footnote-10): git clone, git checkout, git pull, git commit, and git push.

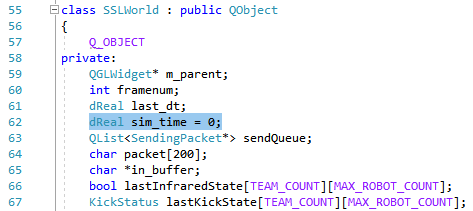
## Grsim modification - accurate speed measurements

As of 10/21/2020, grsim reports wall time instead of simulation time which makes the speeds of the simulated robots dependent on the load of the cpu. To calculate speeds, Robobulls software uses the time stamps received in detection packages (either from vision or grsim). Thus, when using grsim, speed measurements will not be correct.

To solve the issue, grsim needs to be (re)compiled adding / modifying the following 4 lines of code:

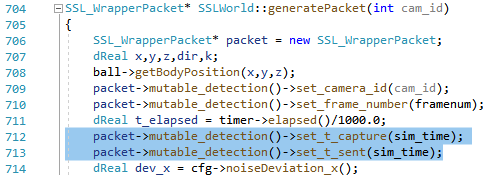
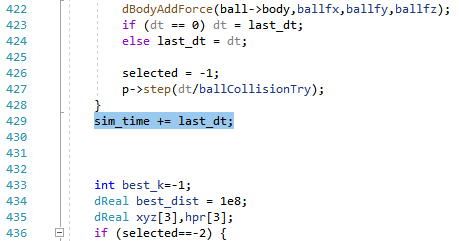
**sslworld.h**

Add line 62: dReal sim\_time = 0;



**sslworld.cpp**

Add line 429: sim\_time += last\_dt;  
Modify lline 712: packet->mutable\_detection()->set\_t\_capture(sim\_time);  
Modify line 713: packet->mutable\_detection()->set\_t\_sent(sim\_time);



# Compiling the code.

## Windows

1. Install Visual Studio[[11]](#footnote-11), during the installations make sure to include “Desktop development with C++” **(is this step needed?)**

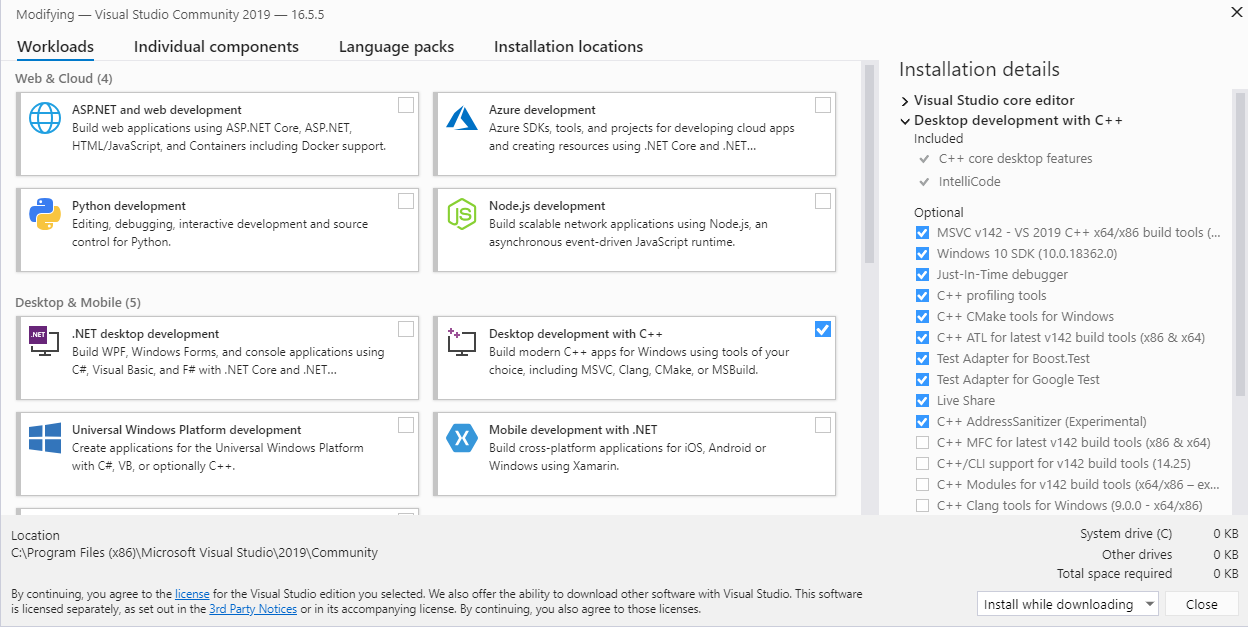
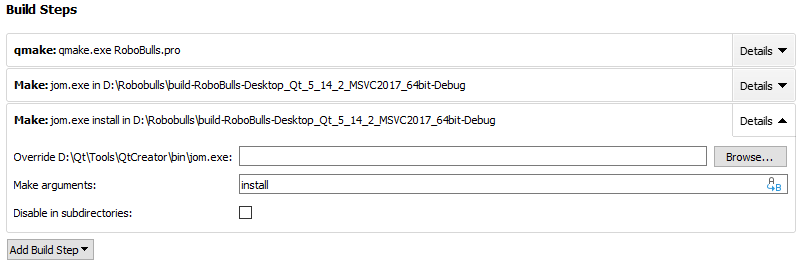
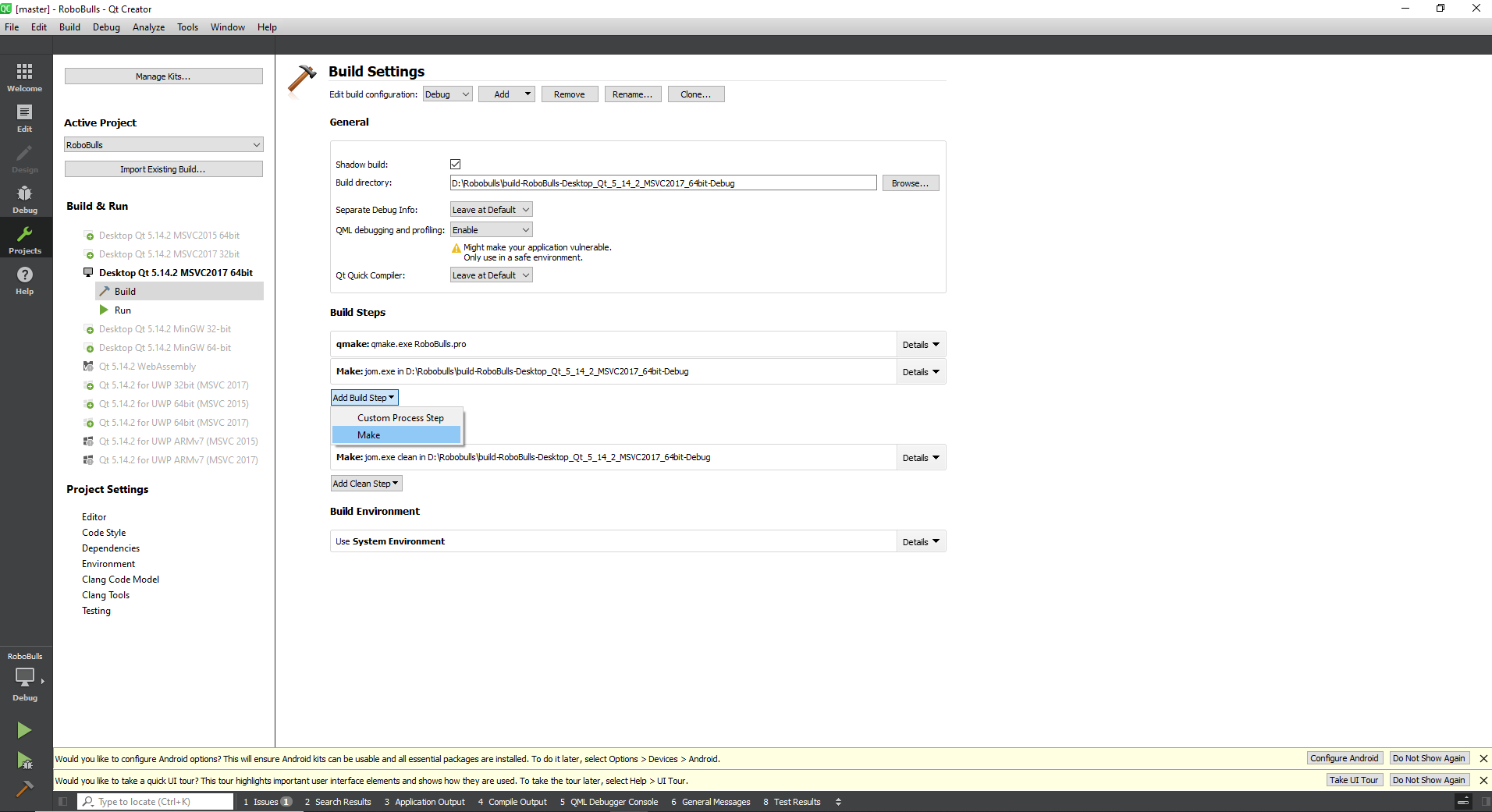
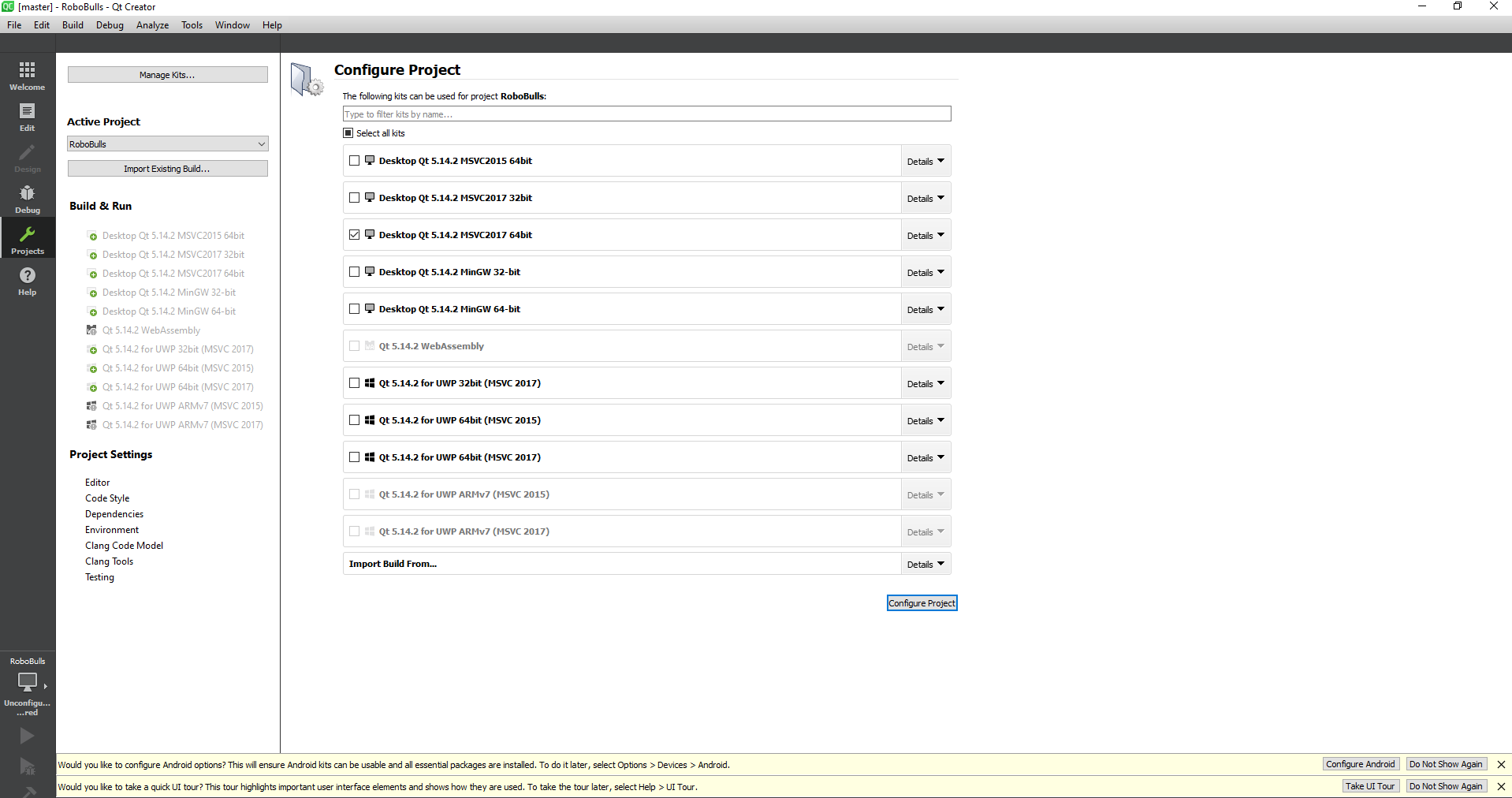


Figure 2 – Windows compilation step I

1. Install Qt creator[[12]](#footnote-12) (there is a free version for the open source community). While installing, choose an adequate Qt version. Use the latest version unless incompatible. As of 05/13/2020 robobulls works with Qt 5.14.2.
2. After the installation, from robobulls main folder, open “RoboBulls.pro” by double clicking it. The first time the project opens Q will request to configure the project. Choose an adequate compiler and then click “Configure Project”. As of 05/13/2020 the software works using “Desktop Qt 5.14.2 MSVC2017 64bit”.
3. Go to the “Projects” page. For each “Build Configuration” add an extra “Make” build step, set its “Make argument” to “install”.



1 (choose adequate)

2

3

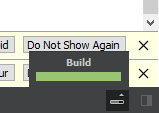
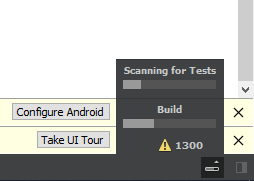
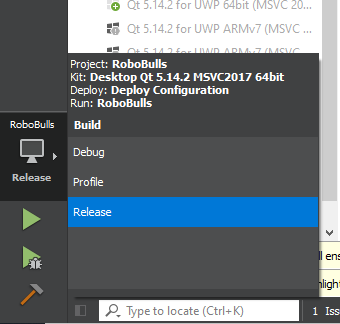
4 (for each config)

5

6 (argument of new step)

Figure 3 – Windows compilation steps III and IV

1. Choose which version to compile (release version in sample image) and then compile. The first time it will take some time, a progress bar can be seen in the lower right corner of the screen. If no errors arise, robobull’s dashboard will load after the compilation is done to the “Projects” page.



7

8

9

10

11

12

Figure 4 – Windows compilation step V.

## Linux

[ SECTION UNDER CONSTRUCTION ]

## MAC

[ SECTION UNDER CONSTRUCTION ]

# Running robobulls software

This section describes how to execute robobulls software, how to configure the initial set up and how to connect to the software provided by ssl.

## Starting the program

After compiling the software, the robobulls executable will be located in the $PROJECT\bin folder along with all required libraries, resources and configuration files. See section 5.2 for documentation on how to configure each configuration file. To run the software, you can either double click the executable (in unix environments make sure the file has ‘execute’ permissions[[13]](#footnote-13)) or run the executable from a terminal as follows.

$ ./robobulls [ CONFIG\_FOLDER ]

Here, CONFIG\_FOLDER is an optional argument that specifies where are the configuration files located. If no argument is used, when the program loads, the program searches the current working directory for a folder named ‘config’ containing all configuration files. Using the optional argument is useful for having multiple sets of configuration files. Currently, it is not possible to modify configurations values once the program has started, feature coming in a future iteration of the program. Meanwhile, to load another configuration you have to restart the program.

When the program loads, it will display both a gui and a terminal. Unless you have already started the vision system (either SSL-vision or grSim) and correctly configured the configuration files, the gui will show an empty field. The terminal will display the contents read from the configuration files. Figure 5 and Figure 6 show the gui and terminal at the start of the program. 

Figure 5 – Robobull’s gui at the start of the program when no vision system is running

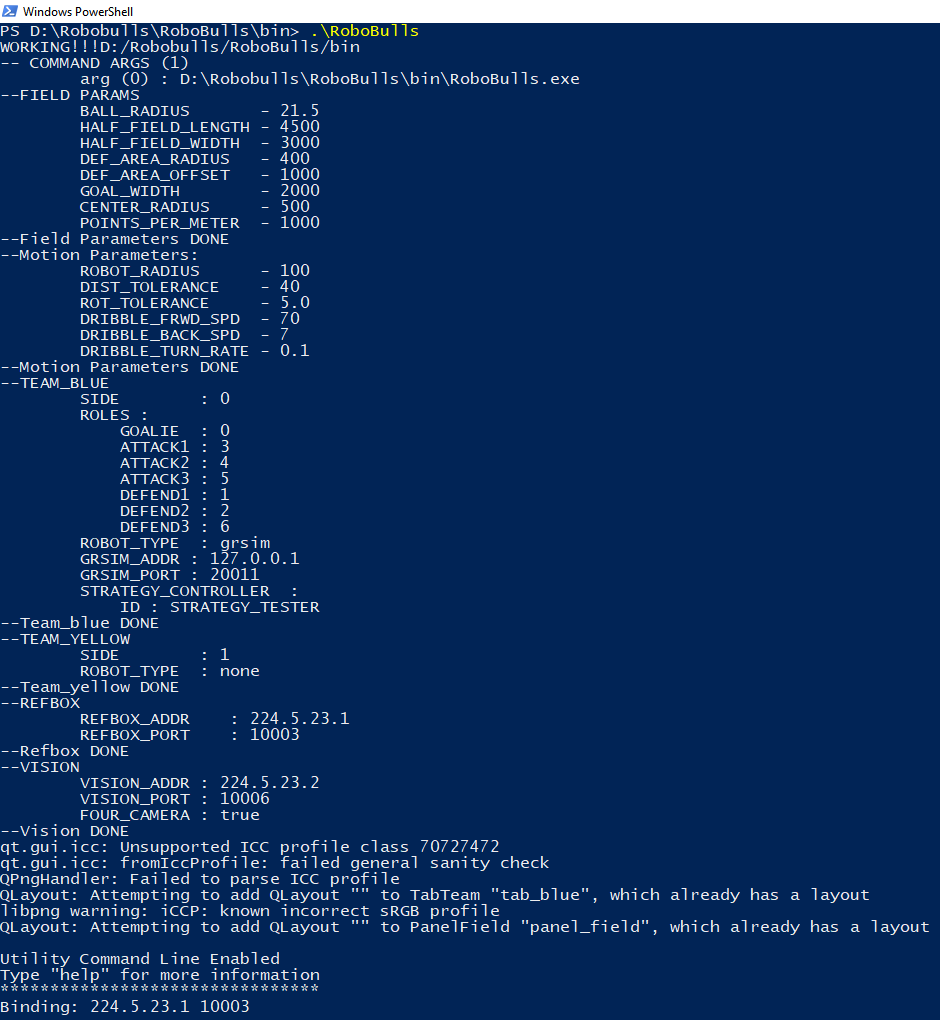


Figure 6 – Robobulls terminal at the start of the program.

## How to set up robobulls configuration files.

As mentioned in section 5.1, when the robobulls software loads, it uses a set of configuration files located at $PROJECT \bin\config. After every successful compilation, these files are replaced by their archetypes found at $PROJECT \config. Thus, any changes made to them gets overwritten. To avoid this issue, you can either modify both the archetypes and the copies or have your configuration files in a different folder and pass the folder as an argument to the program as explained in section 5.1.

**NOTE: If modifying the archetypes, never push your specific changes to the robobulls repository unless you know what you are doing, and you actually mean to modify them in the repository.**

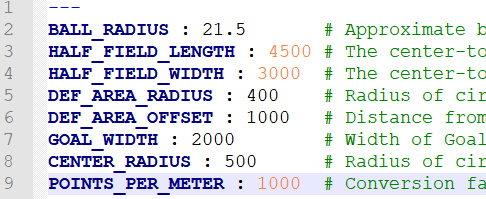
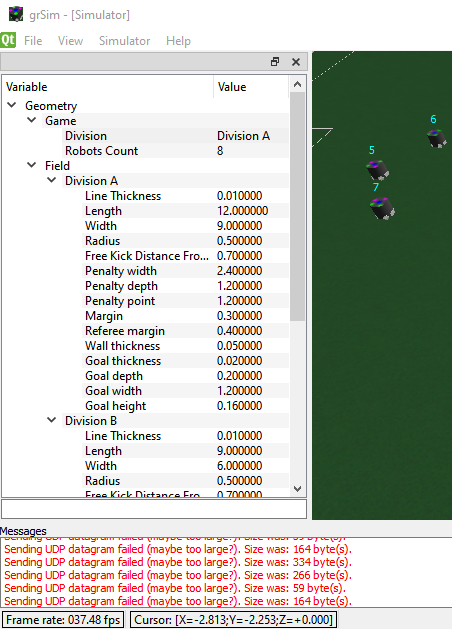
The configuration files are text files formatted using YAML[[14]](#footnote-14). They can be edited using any text editor. YAML is very simple but care should be taken to respect its syntax. YAML is sensitive to the indentation and thus mixing tabs and spaces should be avoided. To find “invisible” indentation errors you can always open the file in an editor (e.g. notepad++) that can display white space symbols such as tabs and spaces.

As of the writing of this document, the robobulls software uses 4 configuration files described in the following sub sections.

### field.yaml

This file defines the dimensions of the field and its markings. Each parameter in the file has a comment explaining what it represents. In general, all measures are represented using millimeters. When using real robots these values should match the dimensions of the real field, and when simulating it should match the dimensions being used by grSim.

Figure 7 - TOP: grSim’s property configuration panel. Highlights in red indicate how to switch between SSL divisions A and B, and where to find their respective field measurements (note that measurements are displayed in meters). BOTTOM: Contents of robobulls field.yaml file (note that measurements are displayed in millimiter).



The measurements of the field in the biorobotics lab can be found pasted on the walls of the lab, or alternatively in the file $PROJECT\config\defaults\field\_biorob.yaml.

If using grSim, the dimensions can be found in the property’s configuration panel found in the left border of grSim’s gui. If the panel is closed you can reopen from the ‘view’ menu. By default, grSim has two possible field configurations, one for SSL division A, and another one for division B (read SSL rules for information about division[[15]](#footnote-15)). The configuration used by grSim can be modified using the property “Geometry->Game->Division”. The measurements con be read/modified under “Geometry->Field->Division X” (where X is A or B). Note that the units in grSim are in meters, and thus they have to be multipled by 1000 when copied to robobulls config files. The following images show grSim’s property configuration panel and the contents of robobulls team.yaml file.

### motion.yaml

This file defines parameters that control robot motion. This file seems likely to become deprecated in the future and should be used by advanced users only. TODO provide more information on this section or deprecate the file (the contents seem more appropriate to be defined on each robot individually).

### comm.yaml

This file defines properties related to communication with vision and the game controller. This file specifies the multicast IP and port numbers of the game controller and the vision system, as well as the number of cameras used by the vision system (currently robobulls only support’s either 2 or 4 camera setups only).

By default, the game controller uses the multicast ip 224.5.23.1 and port 10003. If required for any reason, search the associated software documentation to find how to change its ip and port.

If using simulated robots, grSim uses a 4-camera system and provides vision packages at its default multicast ip address 224.5.23.2 and port 10020. These values can be modified by replacing the properties “Communication->Vision multicast address” and “Communication->Vision multicast port” in grSim’s property configuration panel in its gui.

If using real robots, then the SSL-vision provides vision packages at its default multicast address 224.5.23.2 and port 10006. Similar to grSim, SSL-vision also provides a property configuration panel in which these values can be modified if required. In the biorobotics lab, a 2-camera setup is used.

**NOTE: each program can be run (and usually is run) on a separate machine. Since packages are transmitted via multicast messages this means that all routers connecting the machines must run the multicast protocol (which they usually do not). If routers do not use the protocol (as it is the case for the network on campus), then all computers must be connected via a switch / ethernet.**

### team.yaml

This file defines the settings for both **TEAM\_BLUE** and **TEAM\_YELLOW**. The file comes with sample settings that can be chosen from by commenting/uncommenting different section. Here we describe the file in its entirety.

**FROMAT:**

For each team you must at least specify the tags **SIDE** and **ROBOT\_TYPE**.

* **SIDE** indicates the side of the field that the team is assigned to and it takes values of either 0 or 1 to indicate the negative and positive sides, respectively.
* **ROBOT\_TYPE** indicates the type of robots that the team will be controlling. Currently we require all robots in a team to be of the same type. Possible values for the tag are:

- **none** indicates that the team is not controlled by the robobulls software

- **grsim** indicates that the team uses the robots provided by the simulator grSim

- **yisibot** indicates that the team uses the physical robot yisibot

- **rpi\_2019** indicates the team uses the physical robot rpi\_2019 [UNDER DEVELOPMENT]

For each team with **ROBOT\_TYPE** different than none, we must also specify the following tags:

* **ROB\_COMM** is map of properties which indicate how to communicate with the robot. The properties depend on the type of robot we are connecting to.
* If using robot **grsim**, then **ROB\_COMM** must specify:

- **GRSIM\_ADDR** indicates the ip address of the machine running grsim. If running in the same machine, you can set the value to 127.0.0.1 (localhost).

- **GRSIM\_PORT** indicates the port at which grSim is listening for commands. By default, it is set to 20011 and it can be modified in grSim using the property configuration panel and changing property “Communication->Command listen port”

* If using robot **yisibot**, then **ROB\_COMM** must specify:

- **YISI\_USB\_PORT** indicates the port at which yisi’s transceiver is connected to.

- **YISI\_USB\_FREQUENCY** indicates the frequency channel at which the robot communicates with the transceiver.

For thorough instructions on how to use yisibots, see yisibot documentation in the robot’s doc folder.

* **STRATEGY\_CONTROLLER** is a map of properties that define which strategy controller should be used to control the team. At the very least, the map must include the property **ID** which indicates the controller used to control the team. Other properties will depend on the specific controller. For the details about each controller, see the controller’s documentation in the docs folder. The following is a list of the currently available controllers.

- **ID : STRATEGY\_TESTER** this controller is used to test new strategies. No extra properties are required.

- **ID : NORMAL\_GAME** this controller defines a set of strategies for playing a whole game. No extra properties are required.

- **ID : JOYSTICK** this controller is used when the team robots are controlled by joysticks. It requires property JOY\_TO\_ROBOT\_MAP which is an array of objects containing two properties ROBOT and JOY which indicate which joy id is mapped to which robot id. For an example, see the sample team configuration at the bottom of this section.

* **ROLES** is a map of properties that define which robot will fill each role in the team. The map must assign an integer id (the robot id) for each of the following properties: **GOALIE**, **ATTACK1**, **ATTACK2**, **ATTACK3**, **DEFEND1**, **DEFEND2** and **DEFEND3**.

#### Sample configuratiaon

The following shows a complete sample configuration of the team config file as of the writing of this document:

---

# NOTE 1: hashtags indicate comments

# NOTE 2: YAML is sensitive to indentation

TEAM\_BLUE :

SIDE: 0 # 0 negative side, 1 positive side

# Define the roles of each robot

ROLES:

GOALIE : 0

ATTACK1 : 3

ATTACK2 : 4

ATTACK3 : 5

DEFEND1 : 1

DEFEND2 : 2

DEFEND3 : 6

# ROBOT\_TYPE: yisibot

# ROB\_COMM:

# YISI\_USB\_PORT : /dev/ttyUSB1 # yisi usb port

# YISI\_FREQUENCY: 0

# if robot type == grsim, then must specify:

ROBOT\_TYPE: grsim # alternatives: grsim yisibot rpi\_2019 none

ROB\_COMM:

GRSIM\_ADDR : 127.0.0.1 # ip of grsim

GRSIM\_PORT : 20011 # port of grsim

STRATEGY\_CONTROLLER:

ID: STRATEGY\_TESTER # options: NORMAL\_GAME or STRATEGY\_TESTER or JOYSTICK

# JOY\_TO\_ROBOT\_MAP :

# - ROBOT : 2

# JOY : 0

# - ROBOT : 1

# JOY : 1

# support for rpi\_2019 is work in progress

TEAM\_YELLOW :

SIDE: 1

ROBOT\_TYPE: none #we do not control the team, no extra properties required

## Connecting to SSL software and controlling the robots

### Connecting to SSL

No special steps are required to connect robobulls to the software provided by SSL besides setting the configuring files as described in the previous sections and starting each software manually. After robobulls loads, the program will automatically start listening for vision and referee packages. If packages are received, the robots will appear on robobull’s gui and the game information will be displayed on the “Game Info” panel.

### Controlling robots

When the robots are first loaded, their behavior is overridden by the gui, thus, the controller defined in the configuration file “team.yaml” will not be able to control the robots until the override is released. You can toggle the override of individual robots on and off by clicking the checkmarks next to the robot ids on the gui. Alternatively, you can release/override a whole team at a time by clicking the buttons “Release All Team” and “Override All Team”. Once a robot is released, its behavior will be controlled by the assigned strategy controller. For a description of the expected behavior of each controller see the respective controller’s documentation in the docs folder.

While robots are overridden, you can use the gui to control any robot by first clicking the robot’s icon and then using keys a, s, d and w to move the robot, shift to dribble and space to shoot. For a complete description of how to use the gui see section 6.

### Playing a full game

To have the robots play a full game, configure each team with an adequate controller (such as the controller “NORMAL\_GAME”), run the software and release the behavior of all robots. You will need to use the game controller to act as the referee and control the game.

For more information about the game controller see section 3.

For a list of all available controllers, see section 5.2.4

For detailed information about a specific controller, see the respective controller’s documentation in the docs folder.

### Debugging vision communication issues

If no robots are displayed after loading, verify all the following.

* Make sure that your vision system is running (grSim if simulating or SSL-vision if using real robots).
* Make sure that the file “comm.yaml” was correctly configured to match the vision system as described in section 5.2.3.
* Make sure that there are no routers between the machine running robobulls and the machine running the vision system. Vision packages are transmitted using multicast which by default is not supported by most routers.
* After using grSim for prolonged periods of time, grSim may stops working correctly in which case restart grSim.

### Debugging robot communication issues

To test whether the robobulls software is able to connect to a robot, override the behavior of the robot and use the gui to control the robot as described in section 5.3.2. If you are able to control the robot, the issue will most likely be related to the strategy controller rather than to the connection with the robots themselves.

If you are unable to control the robots using the gui, make sure that the file “team.yaml” is configured correctly as described in section 5.2.4. If it is, then refer to the respective robot’s documentation in the docs folder.

### Debugging game controller communication issues

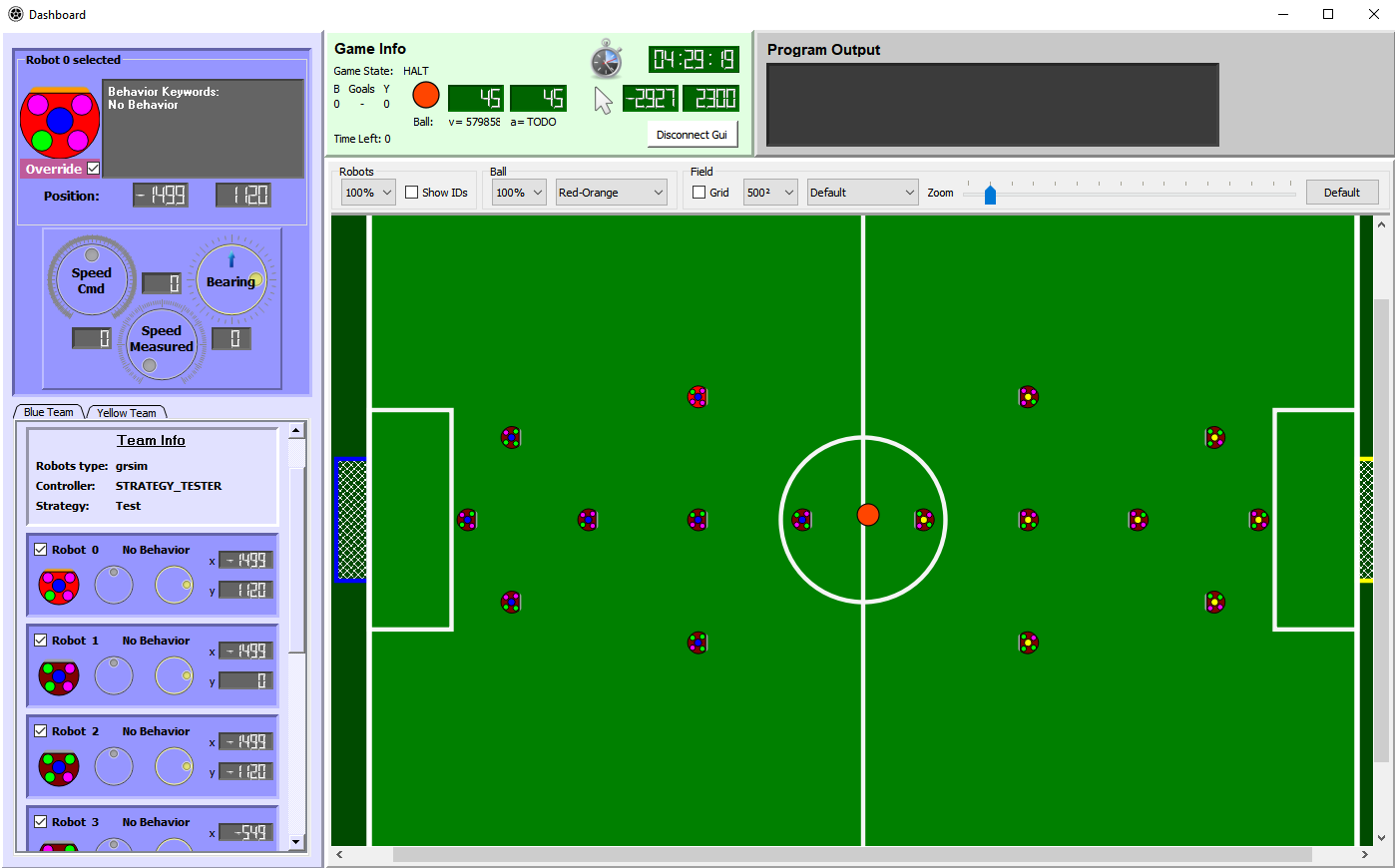
Game controller communication issues will become apparent if the game state is not correctly displayed in the gui’s panel “Game Info”. If so, make sure that the file “comm.yaml” is configured correctly as described in section 5.2.3. Also, note that as of 5/25/2020, the game controller has a bug which requires the game controller to be loaded before robobulls software or vice versa (TODO: verify which of the two is correct).

# GUI

The objective of robobull’s gui is to provide a simple interface to visualize game information at run time and help with the development and debugging of the strategy controllers. This section aims to describes the different parts of the gui along with the tools it provides and the different ways to interact with it.

## Gui components

The gui is divided into five main components shown in Figure 8. The following subsections describe each component.



**Panel Field**

**Panel Selected Robot**

**Panel Output**

**Panel Game Info**

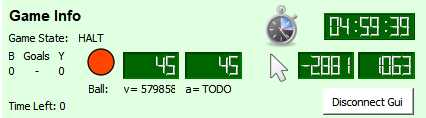
**Panel Teams**

Figure 8 – Robobulls gui and its 5 main panels.

### Panel Output

The output panel is used as mini output terminal to print debug information. Its objective is to be able to see debug data with out to switch back and forth between a terminal.

### Panel Game Info



**Time left in the game**

**Ball x, y coordinates**

**Linear speed  
and acceleration**

**Each team gaols**

**Tme since the start  
of the program**

**Mouse X,Y coordinates**

**Enable/Disable gui button**

The game info panel displays general information about the game which we illustrate in Figure 9. The panel also includes a button to enable and disable the gui. When disabled, all panels become unresponsive and the gui no longer updates until reenabled. **NOTE: some of the panel’s functionality is yet to be implemented.**

Figure 9 – Panel Game Info.

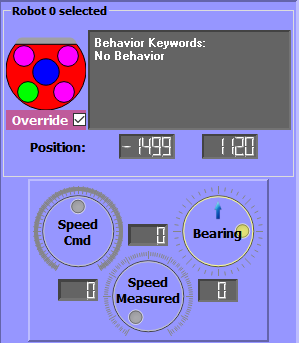
### Panel field

The panel field, illustrated in Figure 9, displays graphic information about the field and the position of the objects in the field. The control bar on top is divided into 3 groups of controls. The first group “Robots” allows to change the size in which the robots are displayed and to show or not robot ids. When showing ids, the added label will also indicate the movement status of the robot (collided, in danger of collision, open) and will mark the robot in possession of the ball as defined by robobulls software (this allows to see whether the algorithms are working fine or not). The second group, “Ball”, allows to change the size and color of the ball. The third group, “Field”, allows to toggle on and off the visualization of a grid in the field, modify the size of the grid, change the colors of the field and the zoom. It also provides a way to return the zoom to its original value.



Figure 10 – Panel Field

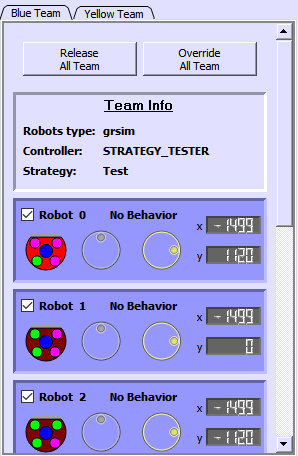
### Panel Selected robot



The panel selected robot, illustrated in Figure 11, shows the information of a selected robot. If no robot is selected, the panel is hidden. The panel provides a small output terminal which prints output messages produced exclusively by the robot; it provides a checkbox for overriding the robot’s behavior (see section 5.3.2); it displays the x, y coordinates in millimeters of the robot in the field and it provides 3 dials with their accompanying lcds either located either above or below the dial. The first dial, “Speed Cmd”, displays the speed at which the robot was commanded to move. The second dial, “Speed Measured”, indicates the actual speed at which the robot is moving, ant the third dial, “Bearing”, indicates the robot’s orientation. **NOTE: currently the second dial does not display correct information.**

Figure 11 – Panel Selected Robot

### Panel Teams



The panel team, illustrated in Figure 12, displays both teams’ information. The panel consists of two tabs, one for the blue team and one for the yellow team. Each tab provides two buttons, one button overrides the control of all robots in the team and the other releases it (see section 5.3.2). Beneath the buttons, a small panel is shown displaying team information including the type of robot that the team is using, the name of its strategy controller and the currently active strategy. Beneath the team information, the panel shows a list of robots in the team that are currently in the field. Each robot displays its id, a check box to override the robot, a label displaying its current behavior, an icon displaying the color marking and two dials displaying the robots current speed and orientation. Note that a highlighted robot icon indicates the robot selected by the user.

Figure 12 – Panel Teams

## User interaction

This section describes all the ways in which a user can interact with the gui.

### Dragging the field

Depending on the level of zoom in use, you may only see a portion of the field. You can move the camera around either by using the scrollbars or by control dragging any empty space in the field.

### Robot selection

The gui allows selecting a robot to display extra information about the robot. To do so, click its icon either in the field or from the panel teams. Alternatively, robots with ids between 0 and 9 can also be selected using the keyboard. To select a robot in the blue team just press the number key corresponding to the robot, otherwise, to select a robot in the yellow team, press control plus the number of the robot. Double clicking a robot’s icon will also select a robot, but it will also lock the camera on the robot (see section 6.2.3).

Once a robot is selected, the robot’s icon will become highlighted and the panel “selected robot” will be displayed with extra information about the robot.

To deselect a robot, you can either press any empty space on the field or just select another one.

### Robot camera tracking

The gui allows to center and lock the camera on a specific robot. To do so, just double click the respective robot’s icon on any of the panels displayed. Doing so will select the robot (see section 6.2.2), and the camera will zoom, center and lock its view on the robot.

To unlock the camera from the robot, either drag the field (see section 6.2.1) or deselect the robot (see section 6.2.2

### Hot keys

The following is a table of the available hot keys and their effects.

|  |  |
| --- | --- |
| Hot Key | Description |
| Num key | Select blue robot with given number |
| Ctrl + Num key | Select yellow robot with given number |
| ` | Select blue robot 0 |
| Ctrl + ` | Select yellow robot 0 |
| Enter | Toggles enable\disable gui |
| Back Space | Toggles enable\disable gui |
| Ctrl + o | Override all robots in the selected team tab |
| Alt + o | Release all robots in the selected team tab |
| O | Toggle selected robot override |
| A | Rotate selected robot to the left |
| S | Move selected robot backward |
| D | Rotate selected robot right |
| W | Move selected robot forward |
| Shift | Dribble the ball with the selected robot |
| Space | Perform kick with selected robot |
| I | Toggle show robot ids on the field |
| G | Toggle show grid in the field |
| Z | Zoom in |
| Ctrl + Z | Zoom out |
| Alt + Z | Default zoom |
| + | Zoom in |
| - | Zoom out |

## Other functions

Besides from the functionalities described here, the gui provides an in interface “gui\_interaface.h” to be used by other modules. The interface allows printing strings to the robot terminals as well as to the output panel. The interface also allows to add shapes (such as points, lines and polygons) to be drawn on the field panel. This allows to display graphical information about the robot such as the path performed or to be performed by the robot, possible points or areas of interest and more. For full details, see the respective header file in the top-level folder of the gui.

# Robobulls debugging terminal

[SECTION UNDER DEVELOPMENT]

# Code

In this section we explain the main concepts in the design of robobulls software. This is an introductory guide, and it is in no way meant to be thorough documentation. This guide is intended to provide a starting point to understand, navigate and extend the code. For full details, refer to the code itself.

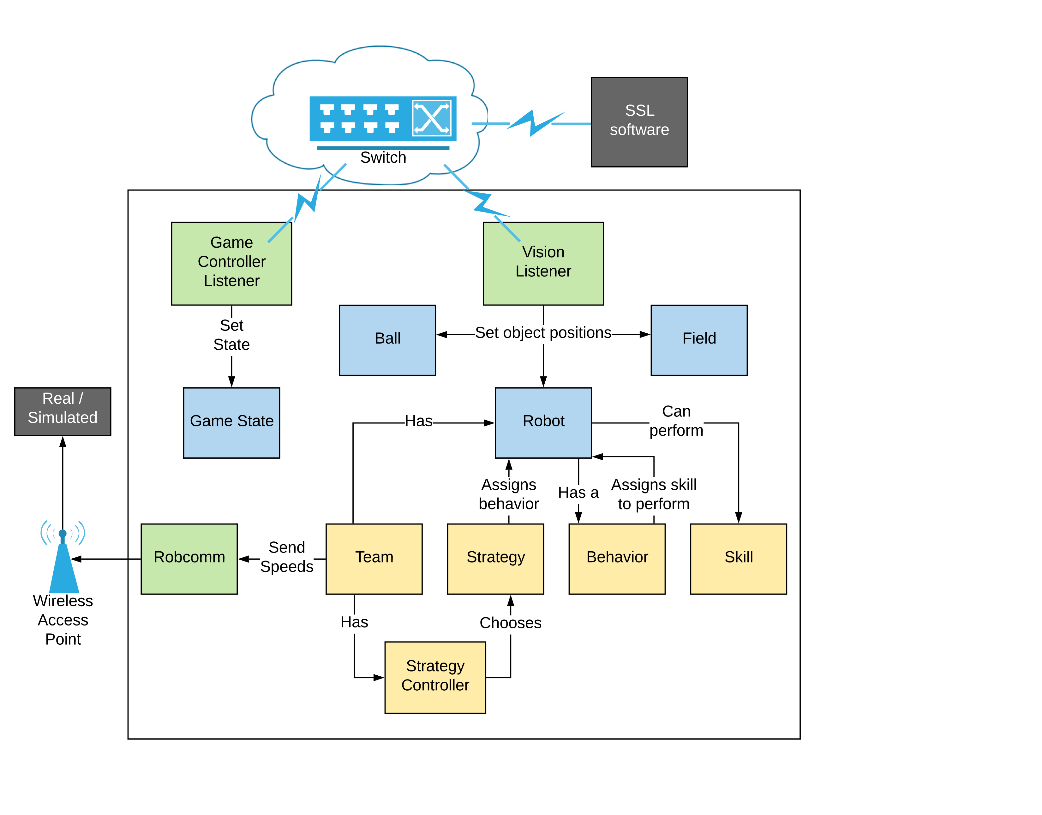
## Skills Required

Using and extending the Robobulls software requires a broad spectrum of skills. The following lists describe some of these requirements. The first list describes the minimum required set of skills to work with the code, while the second describes either recommended skills or skill required for specific tasks.

|  |  |
| --- | --- |
| **Basic** (required skills):   * C++ * Basic YAML scripting * Linear Algebra * Programming in Qt, at least:   + QObjects   + Parent relation between QObjects   + Signal and slots framework | **Advanced** (recommended or task specific skills):   * Mobile Robotics * Multithreading and concurrency * Multithreading in Qt * GUI development * GUI development using Qt * Data structures * Algorithms * Computer Networks   + TCP\IP protocol   + UDP multicasting   + Google Protocol Buffers |

## Main concepts

The robobulls software was originally developed in Linux, using c++ to control a single team of robots playing in Robocup’s SSL league. Since then, the code has been extended to incorporate a gui, support controlling 2 teams at the same time, and to work in all major platforms.



Robobulls   
Software

Figure 13 shows a diagram with the main concepts modelled by robobulls and their relations. Each of the components relates to an implemented class in the software. Gray boxes indicate components outside of robobulls software. Green boxes indicate the components that communicate with external software. Blue boxes indicate basic game elements that store the state of the game and that can be accessed by any component that requires their information. Finally, yellow boxes indicate elements involved in controlling the team of robots.

Figure 13 – Robobulls main concepts.

The main idea behind robobulls, is that the game controller and the vision listeners listen for packages from the software provided by the SSL league. The information received is stored in the classes represented by blue boxes in Figure 13, which can then be accessed by any class using static methods.

The team logic is broken into different levels to allow modularization and reutilization of code. Each team is assigned a strategy controller that chooses strategies to perform based on the current status of the game. Each strategy controls all robots in a team by assigning a behavior to each of robot. Behaviors are high-level task such as “guard point X” or “mark bot Y” that can be split into simpler tasks. A behavior controls only a single robot by assigning it skills to perform. Skills are simple tasks that are usually implemented on top of a robot’s API. Skills could include tasks such as “kick the ball”, “drive to point A”, or “dribble to point B”.

On each cycle, after updating the full team’s control logic, the commands defined for each robot are sent using the class RobComm.

## Folder Structure

The following list represent the files and folder structure used in the code and provides a brief description for each element. The list is not meant to be through but to provide give and idea of the structure of the project.

* **bin** – contains the generated binaries after compiling the code
  + **config** – configuration files generated each time after successful compilation
* **config** – archetype config files
  + **defaults** – predefined default configurations
* **docs** – contains all robobulls documentation
* **libs** – libraries required for compiling the code
* **old** – folder that contains files that are no longer used
* **src** – folder containing robobulls source code
  + **gui** – contains all gui files
    - **data** – folder containg proxies of the game data, these files are the sole contact point between the gui and the rest of the code with the exception of the gui interface.
    - **graphics** – contains QGraphics for drawing on screen
    - **panels** – contains the classes defining the different panels of the gui
    - **style\_sheet** – contains style sheet files that separate the formatting code from the logic code in the gui.
* **gui\_interface.\*** – file providing an interface to use the gui from the rest of the code.
* **main\_window.\*** – defines the main window of the gui
  + **model** – contains the basic classes that model the game (team, ball, field, etc)
  + **parameters** – contains files that separate the parameters from the rest of the code.
  + **robot** – contains all robot related code files
    - **navigation** – contains files that implement robot navigation and perform path planning
    - **robots** – contains code for each individual robot implementation
* **robcomm.\*** - abstract class for defining the communication method for each robot
* **robot.\*** - abstract class for defining robot implementation
  + **ssl** – contains the classes that communicate with the software provided by the SSL league
  + **strategy** – contains the classes that implement all team controlling code
    - **behaviors** – contains all implemented behaviors
    - **controllers** – contains all implemented strategy controllers
    - **skills** - contains all implemented skills
    - **strategies** – contain all implemented strategies that are not specific to a single controller
    - **behavior.\*** - abstract class for defining robot behaviors
    - **skill.\*** - abstract class for defining robot skills
    - **strategy.\*** - abstract class for defining strategies
    - **strategycontroller.\*** - abstract class for defining robot controllers.
  + **Utilities** – contains utility classes used by the rest of the software

## Units of measurements

For consistency, the program uses a standard set of units through-out the code. This prevents having to specify the units for each individual code section and limits the need to convert units only to the classes that communicate with external code. Furthermore, this avoids possible mistakes from confusing units and allows to control different robots through a standardized interface despite differences in specifications.

The following is the list of units used for each dimension:

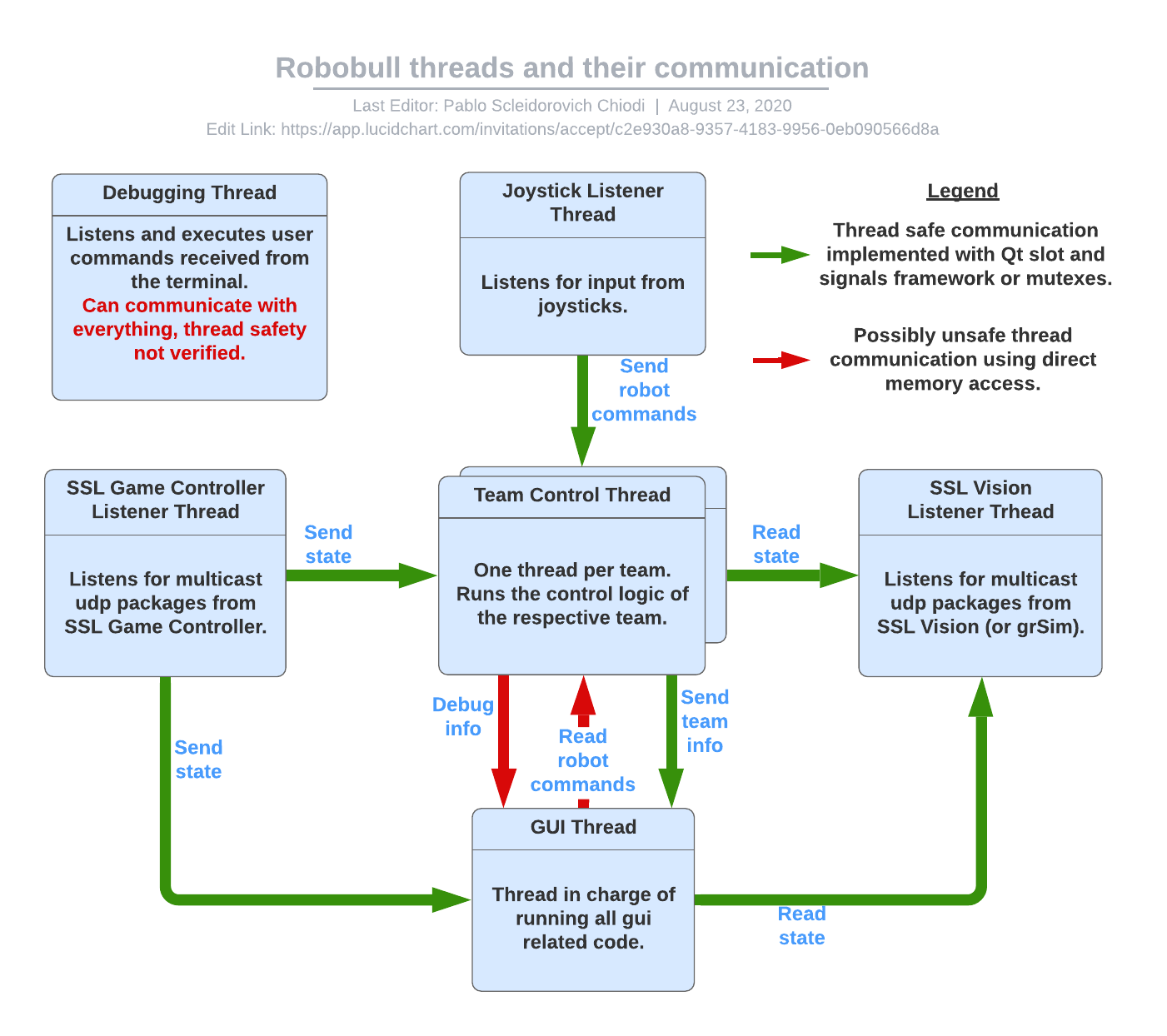
|  |  |
| --- | --- |
| Dimension | Unit |
| Distance | Millimeters (mm) |
| Speed | Millimeters per second (mm/s) |
| Angular speed | Radians per second (rad/s) |
| Kick Power | Millimeters per second (mm/s) [ Not yet enforced ] |
| Dribbling Power | Boolean |

**NOTE: The unit standardization was defined recently, and it has not yet been verified whether all code has been standardized. Do report any issues found.**

## Threads

As of 05/26/2020, the robobulls software uses a total of 4 threads[[16]](#footnote-16), all started from “main.cpp”. Each thread is described in the following subsections. If you are unfamiliar with threads, you should get familiar with threads and concurrency[[17]](#footnote-17) before doing any major changes to the code.

Edit Link: https://app.lucidchart.com/invitations/accept/c2e930a8-9357-4183-9956-0eb090566d8a



**Note**:

To get the information from the SSL Game Controller Listener thread we chose a signaling method (send state) since the game controller state changes very infrequently.

On the other hand, the SSLVisionListener thread processes packages at around 240Hz (60fps x 4 cameras), thus we chose a polling mechanism instead (read state). On each cycle (at most about once every 30 ms), each team controller thread polls the state.

### Debugging thread

The debugging thread opens robobulls debugging terminal. The debugging terminal provides a method to output debugging information, as well as allowing to execute different commands as the code is running. To see the details of how to use the debugging terminal see section 7.

### Game controller listener thread

The game controller listener thread listens for packages from the SSL-Game-Controller software. Once a package is received, it updates the number of goals of each team and then updates the game status with the remaining game time and the received referee command. If the new referee command is different than the last, it also signals the change to each team’s strategy controller.

### Vision listener thread

The vision listener thread listens for packages from the vision system (either from grSim or SSL-vision) and keeps track of the positions and speeds of all robots and the ball.

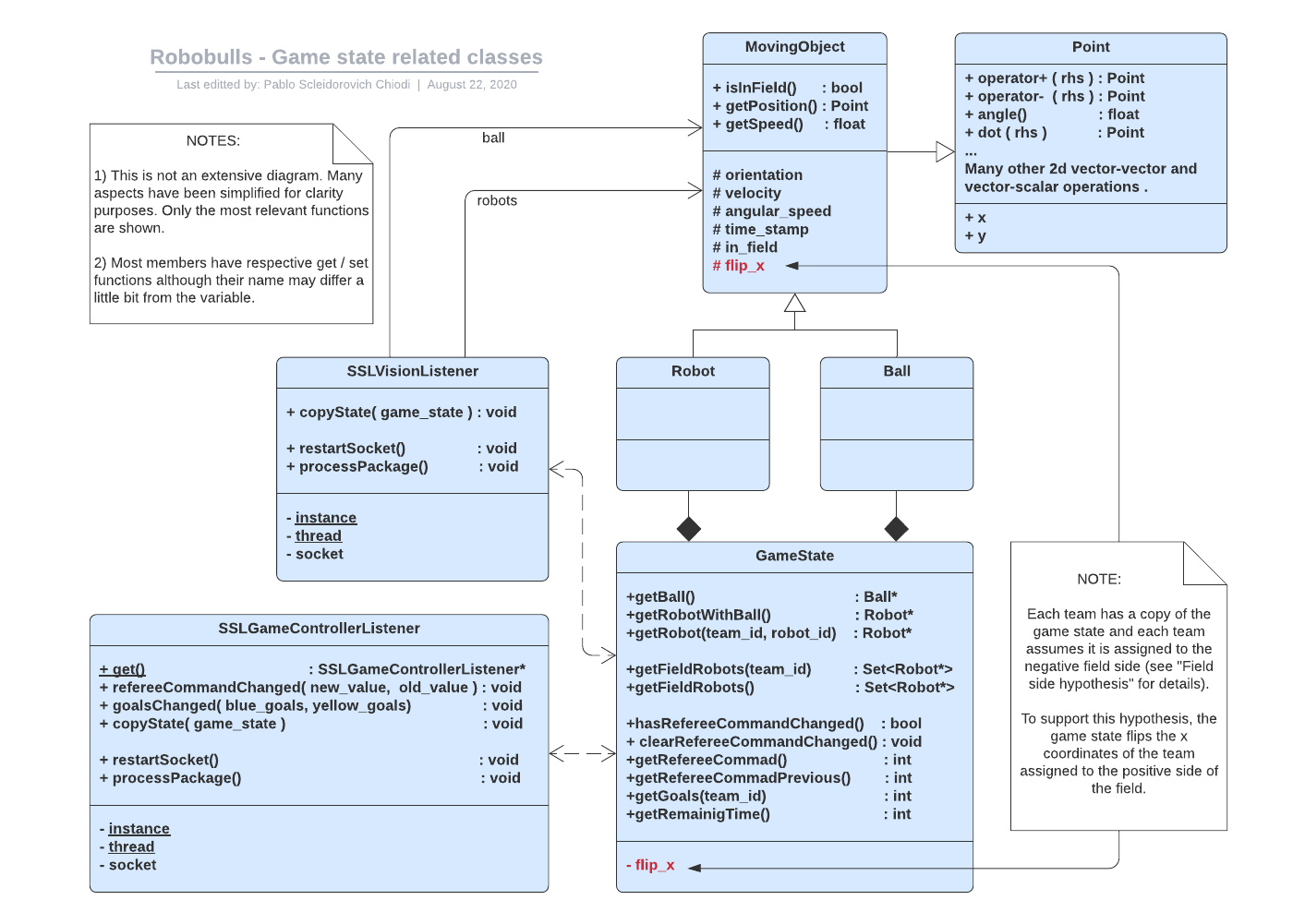
### GUI thread

The gui thread is in charge of processing all gui related code and events. The gui provides a graphical way to visualize the game in real time and provides mechanism for the user to interact with the software (see section 6 for more details). The information displayed on the gui is updated periodically using a timer that generates an event at a fixed frequency.

## Class diagrams & class descriptions

### Game State

Edit link: <https://app.lucidchart.com/invitations/accept/6f3d12d8-8152-4bb0-bc77-acdf90333956>



The state of the game is made available to the teams through the class GameState. In turn, the information of class GameState is set by classes SSLGameControllerListener and SSLVisionListener by processing the UDP multicast messages received from SSL’s provided software (SSL-Game-Controller, SSL-Vision and grSim).

SSLGameControllerListener keeps track of information regarding the overall state of the game such as the last 2 commands received from the referee, each team’s goals, and the remaining time of the game.

On the other hand, SSLVisionListener keeps track of the objects in the field, namely the robots and the ball. For each object, the listener stores its position, orientation, and a stamp of the last time in which the object was observed in the field. From those base values received from SSL’s UDP packages, the listener also calculates and stores the first order derivates, i.e. the velocity and angular speed, and future implementation may add second derivates.

To simplify the code, the game state stores the information of all 32 possible robots (16 for the blue team, and 16 for the yellow team) irrespectively of whether the robots are in the field or not. To differentiate, a Boolean ‘is\_in\_field’ is added to each robot, and special robot sets are maintained to provide access to the robots in the field through functions ‘getFieldRobot( team )’ and ‘getFieldRobots()’. This methodology avoids having to create and delete robots at runtime (along with all associated data structures) and provides place holders to store information of robots that are not yet in the field.

Since each team is controlled from a separate thread, to avoid conflicting memory accesses and to support the ‘team side hypothesis’ (see section 8.6.1.1), each team has a its own independent copy of the game state.

#### Team side hypothesis.

The ‘Team side hypothesis’ is a hypothesis that simplifies team control logic by assuming the team is always assigned to the negative side of the field regardless of whether this is true or not. This assumption removes the need to write special code for differentiating which side is whose, which way is forward / backward and so forth. Instead, the team always assumes that its defense area is in the negative side of the field, and forward points towards the positive x side of the field.

To support this hypothesis, a team’s GameState reflects x coordinates when the team is assigned to the positive side of the field. The team then operates with these coordinates and assigns target velocities to the robots which are flipped back before sending the commands to the real robots. To perform these operations, each instance of class GameState and Robot have an integer ‘flip\_x’ which is used to reflect the x axis. The integer takes the value of 1 when the team is assigned to the negative side of the field and -1 otherwise. Using this integer, the coordinates received by game state from SSL Vision are transformed according to the following equations:

In the equations above , and represent the position and orientation of an object, while , and represent its velocity and angular speed. Non-primed coordinates are the original values provided by SSL-vision, while primed coordinates are the transformed values. Note that when , the primed coordinates are equal to the original set of coordinates. On the other hand, when the primed coordinates are the result of ‘flipping’ (reflecting) the x axis which results in negated x coordinates and angular speeds, with the only non-trivial value being the orientation which results in as demonstrated with the following image:

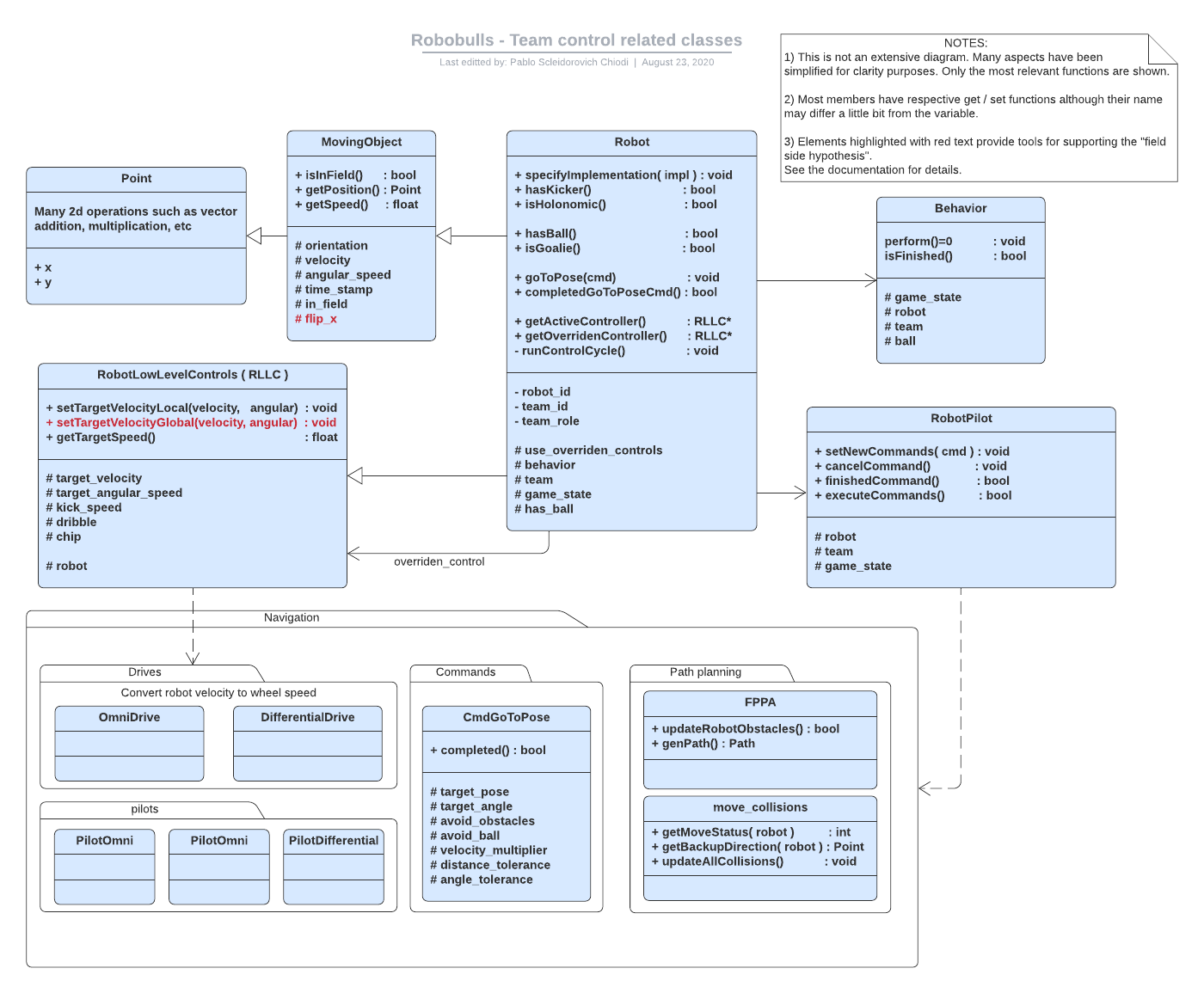


Original  
vector

Reflected  
vector

### Robot Control

Edit Link: <https://app.lucidchart.com/invitations/accept/976a783f-f1bf-4aeb-a568-f6bb1fdb38f7>



In Robobulls software, robots are controlled through class Robot which is independent of specific robot implementations such as drive type, number of wheels, or kicking capabilities. As a result, all robots are controlled in the same way.

The class Robot provides an interface to:

* Query robot information
  + Query the abilities of the robot (such as if it is holonomic or if it has a kicker)
  + Query its state (velocity, orientation, possession of the ball, etc.)
  + Query its role in the team
* Control the robot
  + Control the robot through low level commands
  + Assign a target position for the robot to navigate to
  + Assign high-level behavior

In the following sections, we explore the different ways to control the robot.

#### Low-level control

In the lowest level, robots are controlled by assigning them a target velocity, a target angular speed, and dribbling and kicking speeds. These commands are given to the robot through class RobotLowLevelControls (RLLC for short) which is inherited by the robot. Converting target velocities and speeds into actual motor commands and sending them to the robots is the task of class RobotImplementation, see section XXX for more details.

Each robot has two sets of low-level controls. The first set is the set received by inheriting from class RLLC, which is used to control the robot from the team controller (see section XXX). The second set is provided by a reference to another instance of class RLLC through the robot’s pointer “overridden\_control”. As the name implies, this second set is used to override the controls assigned to the robot by the AI with user assigned controls received from the UI. The Boolean ‘use\_overriden\_controls’ allows toggling between the two sets.

Class RLLC provides two methods for setting the robot’s target velocity: ‘setTargetVelocityLocal’ and ‘setTargetVelocityGlobal’. The first method sets the velocity of the robot based on the robot’s local reference frame, while the second method uses the global reference frame described in section “Team side hypothesis”.

#### Assigning target positions (mid-level control)

In between the low- and high-level control logics, class Robot provides mechanisms to order the robot to move to a given target position and target orientation. This is accomplished by filling an instance of class ‘CmdGoToPose’ and then calling the robot’s method ‘goToPose’. In the command, several parameters can be specified to control different aspects of the motion such as error tolerances and whether the robot should perform obstacle avoidance or not.

After calling method ‘goToPose’, the robot will use its ‘RobotPilot’ instance to set its target speed and velocity on each control cycle. The pilot is in charge of deciding the best way to complete the specified command and provides methods to cancel commands and poll their status. Currently, there are 3 pilot implementations: a dummy pilot which does nothing, a differential pilot for controlling differential robots (as of 08/23/2020 its implementation needs reviewing), and an omni pilot for holonomic robots. To perform its tasks a pilot has access to the robot it controls, its team, the game state, the ball and a navigation package with tools for performing path planning and polling robot collision status among other things.

**IMPORTANT NOTE**: since pilots make use of the ‘field size hypothesis’, all coordinates should use the transformed reference frame as explained in section 8.6.1.1. While usually nothing needs to be done since all coordinates already use the transformed reference frame, if manually assigning a position to go to, make sure the coordinates use this reference frame, otherwise the robot will go to the reflected position.

#### High-level control

High level control of robots is done by assigning behaviors to the robots. A behavior uses lower-level commands to control the robot based on the status of the game. To simplify information access, each behavior has a pointer to the robot it controls, its team, the ball, and the game state.

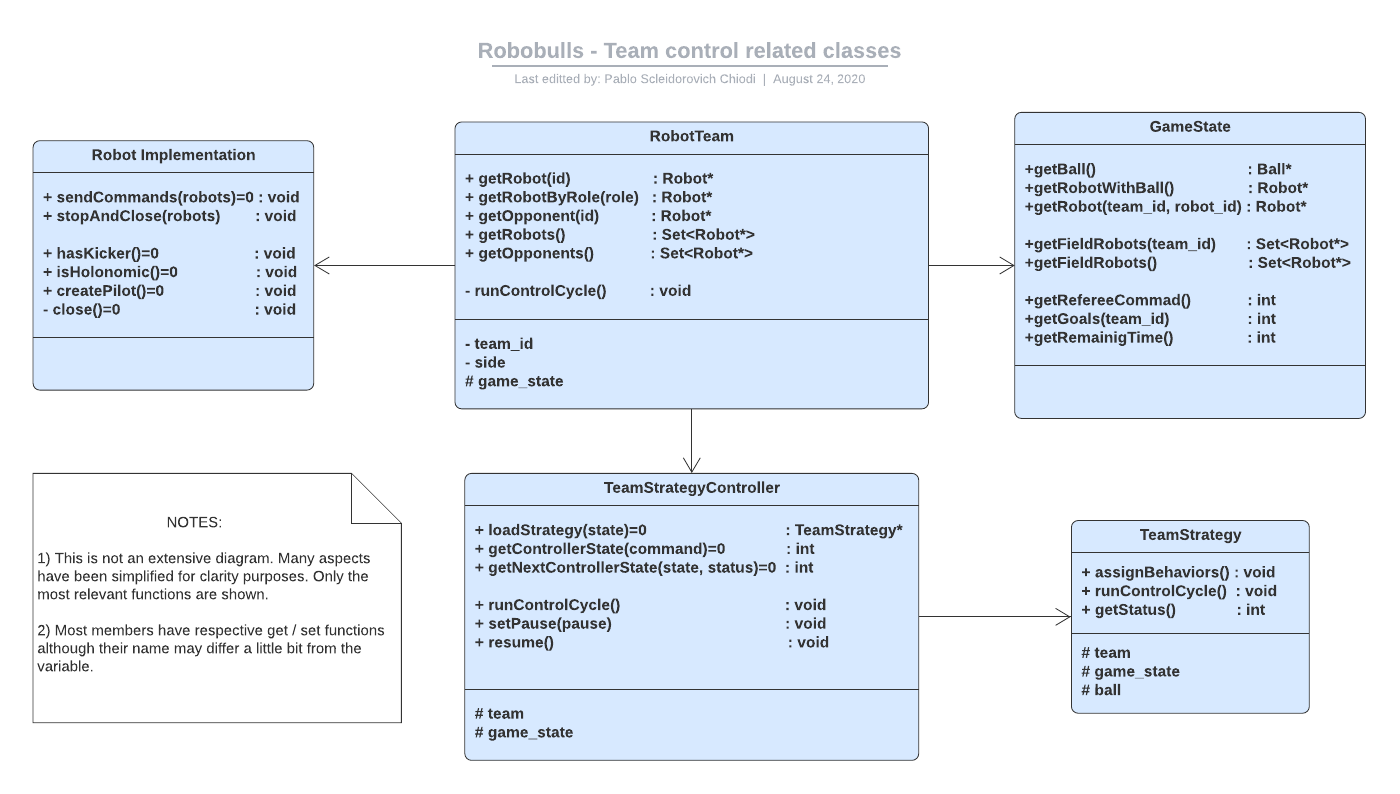
**IMPORTANT NOTE**: since behaviors make use of the ‘field size hypothesis’, all coordinates should use the transformed reference frame as explained in section 8.6.1.1. While usually nothing needs to be done since all coordinates already use the transformed reference frame, if manually assigning a position to go to, make sure the coordinates use this reference frame, otherwise the robot will go to the reflected position.

#### Robot control cycle

On each robot control cycle, a robot first executes its behavior control cycle and then its pilot control cycle. Thus, if a behavior wants to control the velocity of the robot through low-level commands, the current pilot action should be cancelled first, otherwise it risks being overwritten by the pilot.

### Team Control

Edit Link: <https://app.lucidchart.com/invitations/accept/acad4e9f-a911-4ae3-84e4-25fd80fd80e4>



In robobulls, each team is represented by one instance of class RobotTeam that provides access to the robots in the team (ether by id or by role), to the opponent robots, and to its own private copy of the game state.

The team also has a pointer to an instance of class RobotImplementation which defines the capabilities of the robots in the team and provides methods for creating pilots for the robots and communicating with them. In particular, the class is in charge of sending control messages to the real / simulated robots based on the commands stored in the instances of class Robot.

To control the teams, each team has its own team controller which runs in its own independent thread. The details of this mechanism are discussed in the following sections.

#### TeamStrategyController

A team controller is implemented as an instance of class TeamStrategyController, which is responsible for choosing the team strategy based on the current game state. The controller is basically a state machine, whose inputs are the commands send by SSL’s game controller along with the status of the current strategy. Based on those inputs, the controller switches its state and loads a new strategy accordingly.

To facilitate access to the game information, each controller has a pointer to the team it controls and to the game state.

As of 8/23/2020, there are 3 controllers implemented: ‘SController\_Joystick’ which controls robots through joysticks, ‘SControllerNone’ which implements a dummy controller, ‘SControllerStrategyTester’ which is meant to test strategies, and ‘SControllerNormalGame’ which implements the only current game controller.

#### TeamStrategy

A team strategy is implemented as an instance of class TeamStrategy which is tasked with controlling all robots in the team to achieve the respective strategy goal. Normally, a strategy controls robots by assigning a behavior to each robot based on the state of the game, but it can also be used to control the robots through lower level commands.

To facilitate access to the game information, each strategy has a pointer to the team it controls, the ball, and the game state.

#### Control Cycle

On each team’s control cycle the team first updates its copy of the game state and then runs its controller’s control cycle, then the control cycle of each robot, and finally sends the commands to the real robots through its instance of class RobotImplementation.

During the controller’s control cycle, the controller first updates its internal state based on the current game state, the commands received from SSL’s game controller and the current strategy’s status. If the state of the controller changed, a new strategy is loaded accordingly. Lastly, the strategy’s control cycle is executed which assigns commands of any level to the robots.

#### Starting the AI

When Robobulls software starts, all robots are overridden, and the override needs to be released through the UI before the AI can take control of the robots.

## Adding code

### Adding support for new robots

To add support for a new robot, you have to do all the following:

1. Create a new folder titled in the same way as the robot both under $PROJECT\src\robot\robots and $PROJECT\docs\robots.
2. Generate 2 new classes by extending class Robot and class RobComm and place them in the source folder created in step 1.
3. Add a method to load your robots from a YAML file by modify functions loadRobot and loadRobComm in files $PROJECT\srs\robot\robot.cpp and $PROJECT\srs\robot\robcomm.cpp respectively.
4. Update section 5.2.4 of this document.
5. Add any extra documentation on how to use the robot in document folder created in step 1.

### Adding strategy controllers

To create a new strategy controller, you have to do all the following:

1. Create a new folder for the new controller both under $PROJECT\strategy\controllers and $PROJECT\docs\robots.
2. Generate a new class by extending class StrategyController and place it in the source folder created in step 1.
3. Add a method to load your controller from a YAML file by modify function loadController in file $PROJECT\srs\strategy\strategycontroller.cpp.
4. Update section 5.2.4 of this document.
5. Add any extra documentation on how to use the controller in the document folder created in step 1.

### Adding strategies

If you are developing a strategy it is advisable to first test your strategy by modifying the strategy of the strategy tester controller found in folder $PROJECT\srs\strategy\controllers\strategy\_tester.

When done, follow similar steps as the previous section but place your new strategy either in the strategies folder of an adequate controller (if it is specific to a single controller) or add it in under $PROJECT\srs\strategy\strategies if it is for general use.

## SSL communication

SECTION UNDER CONSTRUCTION – PROVIDE A DESCRIPTION OF HOW THE COMPILING OF THE PROTO FILES IS DONE

# TODO List

The following sections describe things to do for new member as well as things that need to be resolved in the robobulls project.

## Tasks for new members

New team members or users of the robobulls software should do all the following::

1. Install software
2. Understand the dependency with SSL’s software

* SSL vision sends udp multicast packages
* SSL Game controller sends udp multicast packages
* Must configure robobulls to listen for multicast packages on the same multicast ip and port

1. Understand how to configure and run the software

* Edit configuration files
* Then start software
* To allow the AI to take over the control of the robots click release team (do I for each team controlled by the ui)

1. Know in which units is the system implemented
2. Understand how to use the GUI
3. Understand how software works:
   * + Threads running and what is communicated between threads
4. Understand the project folder structure
5. Understand team and robot controlling system
   * + High level control:
       - Class StrategyController and the implemented controllers
       - Class Strategy and the implemented strategies for each controller
       - Class Behavior and the implemented behaviors
     + Low level control
       - Class Robot
       - Class LowLevelController
       - RobotPilot

## Documentation

1. Finish writing this document
2. Document each robot
3. Document each strategy controller
4. Document each strategy
5. Document each behavior

## Main code

1. Fix performance issues. When robobulls and grSim run together, grSim’s fps decreases drastically.
2. Fix the issue described in section 8.5.3, that is, make each team controller run on a different thread.
3. Verify all code is thread safe. This issue requires experience with multithreading so it should be dealt by an advanced programmer.
4. Verify all code uses the set of units as defined in section 8.4
5. Verify each behavior is working properly.
6. Verify each strategy is working properly.
7. Review the path planning algorithm (has issues)
8. Test full game
9. Find current bottlenecks
10. Add tool to send ssl-commands from within robobulls for testing software

## GUI

1. Modify the size of the robot and the ball in the display so that when they are drawn at 100% they are correctly drawn to scale.
2. Make all the code thread safe. See related issue 3 in main code.
3. Use png images instead of graphics for the robot icons
4. Improve the performance of the gui (drawing should be done more efficiently).
5. Improve the colors of the gui and make them consistent
6. Finish implementing the Game info panel
7. Fix, gui display of the field, field scale is not correct
8. Further develop the gui. Sample idea:

* Add time series plots and other plotting tools
* Add tools to modify the configuration properties on the fly, when loading display a window requesting to verify the data before continuing.
* Add a method to control the state of the game without having to use ssl-game-controller

## grsim

1. grsim has very low frame rate when running side to side with robobulls software. Is this expected behavior of grsim or does either program have performance issues?
2. Add tool to load specific scenarios in grsim from robobulls software

1. Robocup site - <https://www.robocup.org/> [↑](#footnote-ref-1)
2. SSL software - <https://ssl.robocup.org/league-software/> [↑](#footnote-ref-2)
3. SSL-game-controller (binaries available) - <https://github.com/RoboCup-SSL/ssl-game-controller/releases> [↑](#footnote-ref-3)
4. SSL-vision (requires compiling, already available at the lab) - <https://github.com/RoboCup-SSL/ssl-vision-client/releases> [↑](#footnote-ref-4)
5. In the biorobotics lab, SSL-vision is installed on pc “Vision” on the pass-free user “Vision” (see tags on top of monitors). [↑](#footnote-ref-5)
6. grSim (requires compiling) - <https://github.com/RoboCup-SSL/grSim> [↑](#footnote-ref-6)
7. SSL-vision-client (binaries available) - <https://github.com/RoboCup-SSL/ssl-vision-client/releases> [↑](#footnote-ref-7)
8. SSL contact info - <https://ssl.robocup.org/contact/> [↑](#footnote-ref-8)
9. Git website - <https://git-scm.com/> [↑](#footnote-ref-9)
10. Basic git tutorial - <https://rogerdudler.github.io/git-guide/> [↑](#footnote-ref-10)
11. Visual studio - <https://visualstudio.microsoft.com/> [↑](#footnote-ref-11)
12. Qt Creator - <https://www.qt.io/download> [↑](#footnote-ref-12)
13. Tutorial on unix file permissions - <https://www.softwaretestinghelp.com/unix-permissions/> [↑](#footnote-ref-13)
14. YAML basics tutorial - <https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html> [↑](#footnote-ref-14)
15. SSL rules - <https://ssl.robocup.org/rules/> [↑](#footnote-ref-15)
16. Threads - <https://en.wikipedia.org/wiki/Thread_(computing)> [↑](#footnote-ref-16)
17. Beginner’s guide to concurrent programming - <https://www.toptal.com/software/introduction-to-concurrent-programming> [↑](#footnote-ref-17)