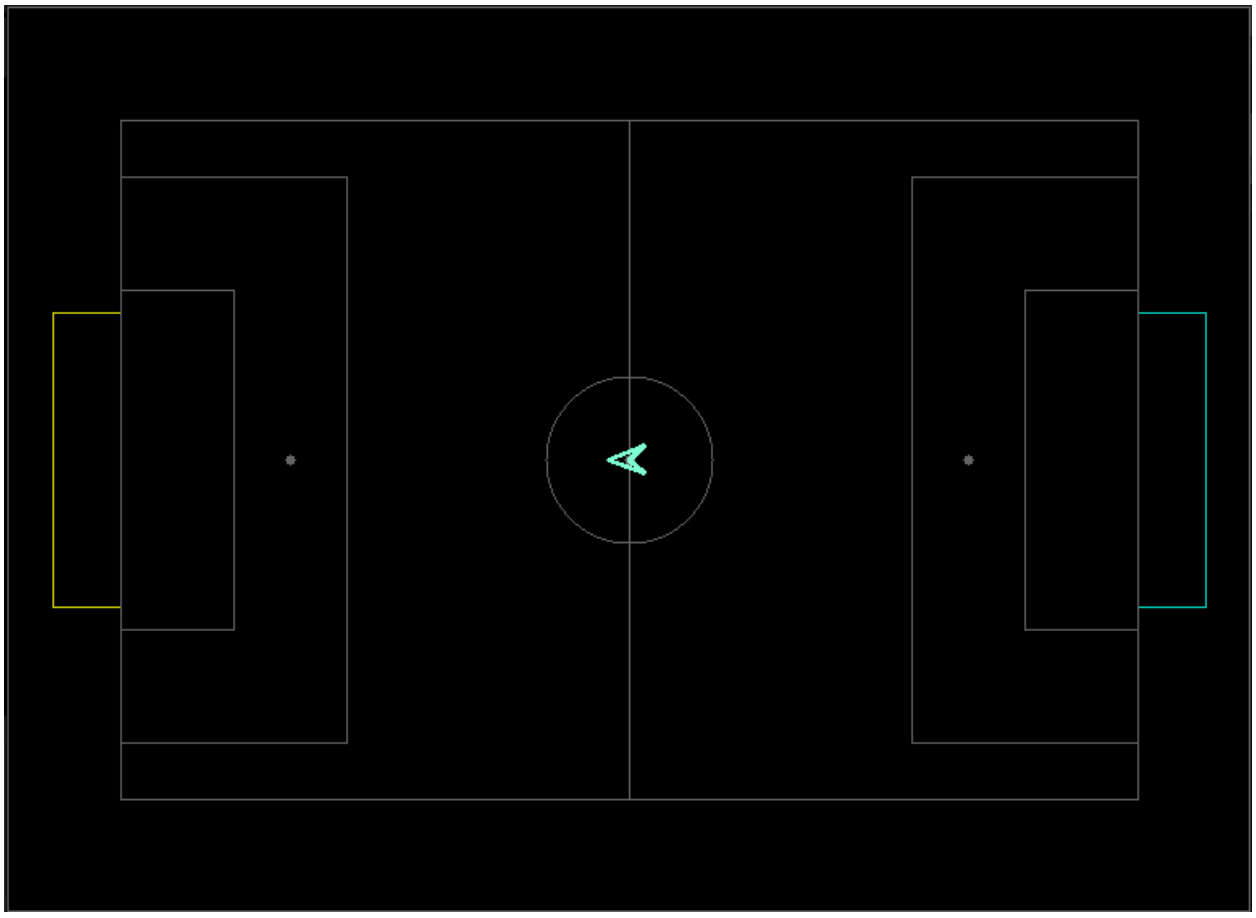
.:: Robot World Model ::.



Authored By : Mahdi Zeinali

Developers : Mahdi Zeinali & Ramtin Kosari

Authored Mail : Zeinali.mahdy@gmail.com

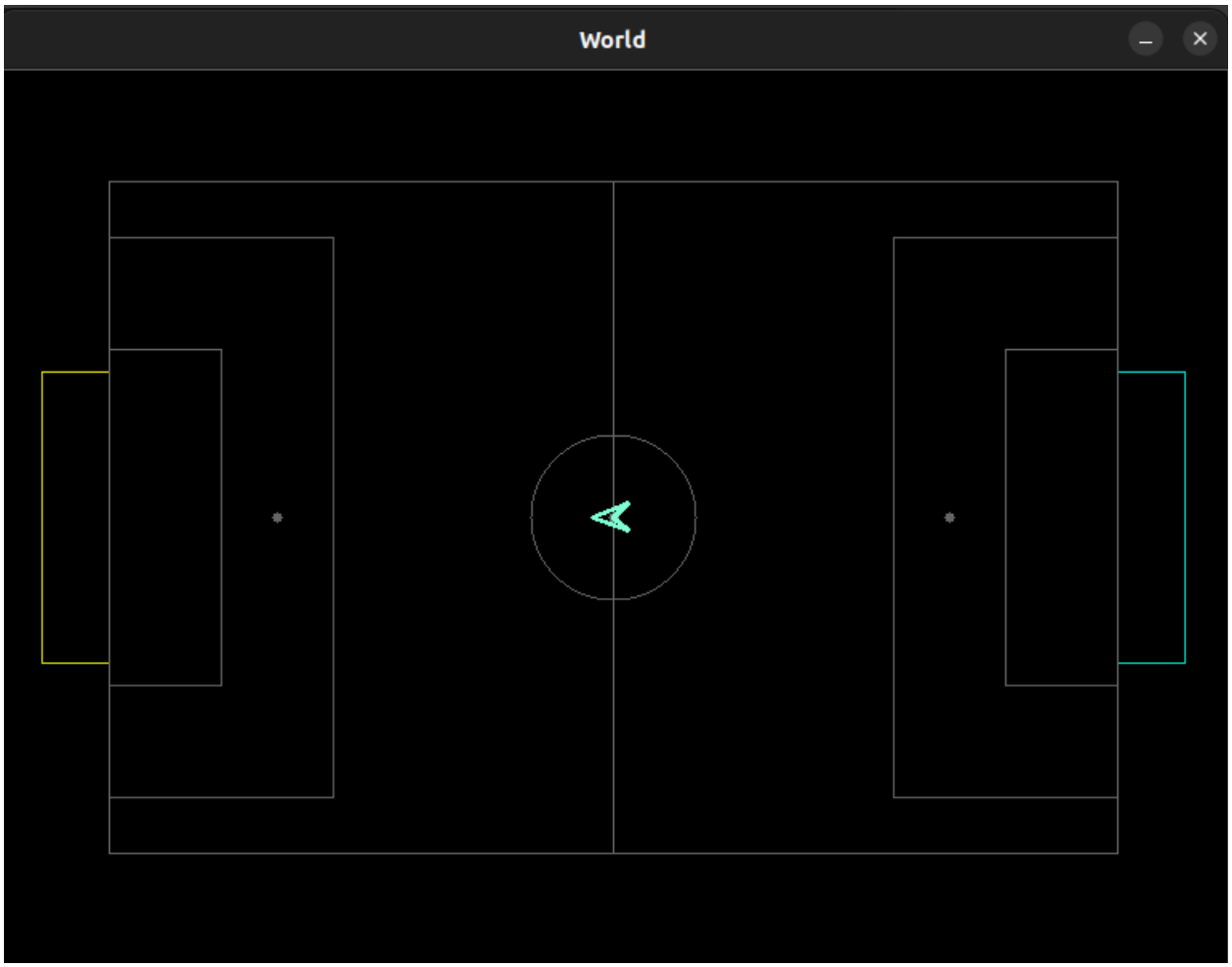
MRL-HSL Mail : mrlhslsoftware@gmail.com

Table of Contents :

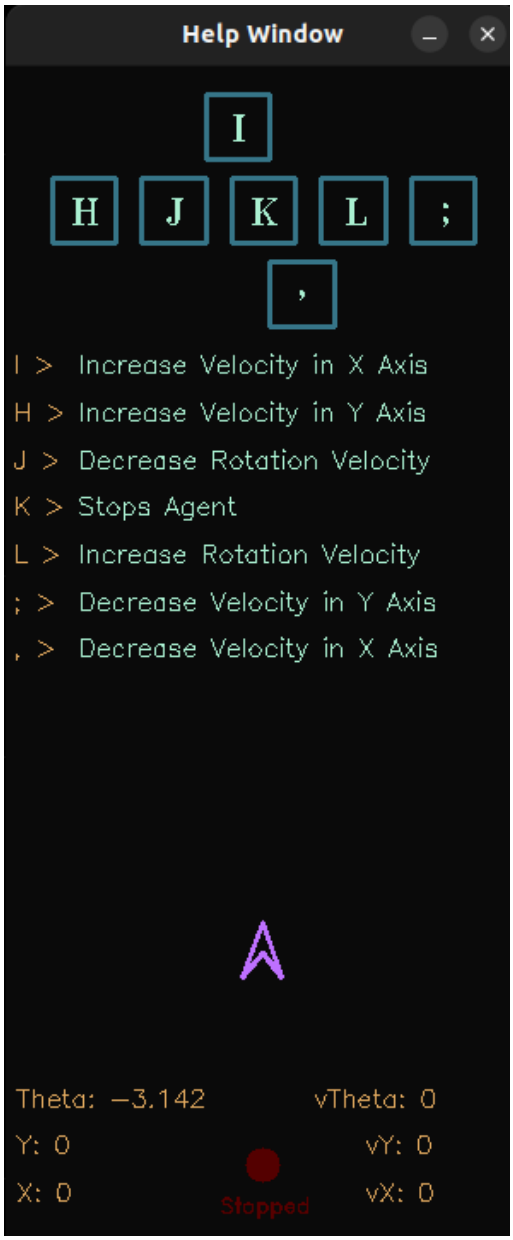
What’s World Model

• Introduction

World model is a dynamic simulation of soccer robots in a supposed soccer field. There are two main windows to represent the real world of robot and the modeling part . The difference between these parts is explained in Concepts . The field has been created by the [Humanoid Robot League Laws 2022] and the modeling process is ongoing with just an agent that is our soccer robot !!



It is obvious in the picture that the agent recognizes the left side as its goal and the right one as the opponent's goal . The outermost region of the field, considered as moving border which will limit the agent to move more than absolut field. The robot is controllable by a user with keyboard and mouse events. In order to get more information about how to use the program, refer to the Getting start section.

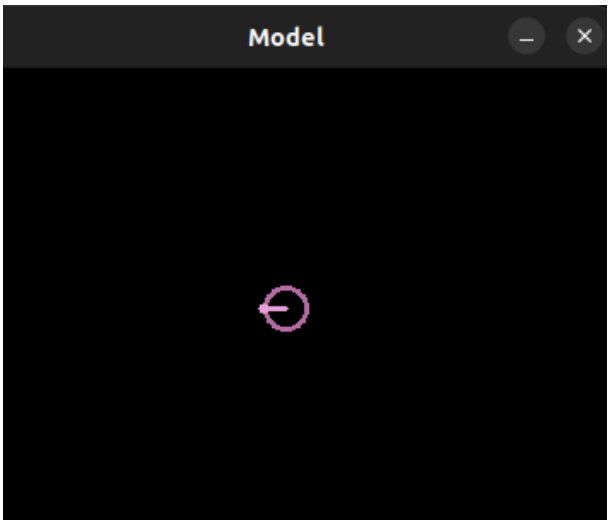


In addition to the World window and Model window which is going to be explained, we have created an extra window to represent the agent status and the location of it inside the field , also the vector of velocity originally based on m/s.

At the top of the Help Window, the keys which are going to be used, have been declared.

It's possible to drag and drop the agent inside the World window by scrolling the mouse; but the important note of this procedure is that this event won't effect on Model window anyway !!

And finally the Model window that its task is displaying the modeling process of the world window by receiving data from the agent. In order to distinguish Model shape, it has been drawn with different shape that you can see and also there is no any field or border in this case .



• Modeling Concept

World window is a natural simulation of a robot, it means that every movement or rotation inside the field is totally sensible. When a normal condition is stable, the agent moves around the soccer field and rotates from 0 to 180 degrees based on angle mode.

Agent's velocity can increase or decrease with the keyboard by user, Also datas will be upgraded and be shown on the Help window. In this situation, Model window sets its agent model's velocity based on the

original agent in the World window; after that, displays a simple movement and rotation inside a none-border field.

Everything will be right until we involve the mouse events; Actually in the real world of robots, it's possible to catch and pick up a robot from a field even though that robot is moving; but the exact point of this rule is, every humanoid soccer robots upgrade their data about surround and their location while they are doing this events by themselves; So it means that when we lift a robot up and put that in another point, or maybe in other rotation, robot's data will no longer upgrade after its last position; therefor if we put it on somewhere , the robot is going to be confused and probably stop working. Don't forget that the model agent has the latest velocity data and keeps continuing its movement without considering robot changes, in fact it might move out of the window and leave an empty field for us.

The goal of this project is representing humanoid robots working whether involving mouse or not in a Simulation environment.

Getting Start with Program

Primary Dependencies

This project has been completely developed on Linux-64 - Ubuntu 22 distribution.notice that previous versions have a bit of a problem.
All the codes are based on C++ language and we have hardly tried to write the codes with simple syntax to make it easier for understanding and help beginners to develop it much better.
In addition to the default libraries of c++, the program is powered by OpenCV v4; As matter of fact, all the elements inside the windows have been designed with OpenCV and there is no any foreign pattern.
In order to use the program , you have to install these dependencies correctly and then try the make rule.

- **Ubuntu 22 LTS version download link**
- **C++ is basically installed on Ubuntu**
- **Opencv 4 installing orientation on Ubuntu**

• How to install & Run

After installing dependencies at first you have to clone the source code from MRL github repository which you can find in the resource section. Or maybe you decide to try it in the Ubuntu terminal :

Cloning the project :

```
$ git clone https://github.com/mrl-hsl/Soccer-Robot-Playground.git
```

Enter the exact directory :

```
$ cd Soccer-Robot-Playground.git
```

Header files are existed here to compile the program :

```
$ make
```

Suddenly after installing the program, windows will automatically open and you can control the agent. please refer to the control orientation.

Maybe you want to delete compiled files :

```
$ make clean
```

Every time if you change the source code or develop it , you have make install again the program but if you want to just run the code , try this order :

```
$ ./main
```

• Control Orientation

1 - Increase the robot movement velocity or start moving

2 - Stop robot movement and rotation on every point

3 - Decrease the robot movement velocity or stop moving

4 - Increase the robot rotation velocity or start rotation

5 - Decrease the robot rotation velocity or stop rotation

6 - Moving up on X axis

7 - Moving down on X axis

8 - Reset robot position and start from base

1 - To drag the agent in the World window

3 - Change mouse event mode for rotation

2 - Rotate the agent from right

1 - Rotate the agent from left

Program Section

• Headers Explanations

When you clone the project, you will face many kinds of files which may cause you don’t know what they are doing while the program is running ; in this scope let’s probe all the header files !!

List of Headers

Field.hpp	**Field.cpp**
Robot.hpp	**Robot.cpp**
Model.hpp	**Model.cpp**
World.hpp	**World.cpp**
HelpWindow.hpp	**HelpWindow.cpp**
Config.hpp	**Main.cpp**

At least there is a Makefile besides project sources that help users and developers to compile the program much easier and also avoid probable problems when the terminal is trying to compile all the headers at the same time with each other.

Before starting the headers description it's necessary to mention that inside the hpp files , the exact class of madul has been declared and encapsulated which contain its functions and private variables in it. Therefore at the top of every cpp file , we must include the name of the hpp file to have the correct connection.

★ Config

Actually this is the main header file which has been declared and included inside all other headers; some variables must be declared in global form to have been accessed everywhere . config.hpp consists of them with some conditions that will be evaluated .

In addition to the variables we have to include all default libraries and declare namespaces inside this header.

C++ language makes it possible for us to create a global variable without quantification at first with a simple order type which is (extern); extern exact location is behind the data types of variables and we can quantify them inside other existing header files.

For instance : `extern double robotSize;`

★ Field

Field section has a specified duty to create the soccer field with Opencv functions and pre-specified padding scale.

```
#ifndef FIELD_HPP
#define FIELD_HPP

#include "Configs.hpp"


//-- Field Class
class Field {
    private:
        Mat Ground;
    public:
        Field();
        void fieldCreate();
        Mat Access();
};

#endif // FIELD_HPP
```

OpenCV generates a picture with a matrix; because the computer can only understand the one-zero and the monitor just sets the brightness with the counts of the image matrix in each pixel.

```
double windowWidth = (fieldWidth + 2 * fieldPadding);
double windowLength = (fieldLength + 2 * fieldPadding);
```

This part of code is going to make fit scale of the window; Based on the rules it must be 9 * 6 so we have changed the pixel size to the meter size.

```
//-- Constructor to Set Default Values
Field::Field(){
    double Scale = modelScale * 12;
    if (Scale >= 2500){
        fontSize = 1.7;
        lineSize = 4;
    }
    else if (Scale > 1550 && Scale < 2500){
        fontSize = 1.2;
        lineSize = 4;
    }
    else{
        fontSize = 0.6;
        lineSize = 1;
    }
}
```

As it's obvious , Field() is a constructor of the field class that sets the default values to the global variables which we have declared out of class.

```
void Field::fieldCreate(){
    //-- Creates the Window
    Ground = Mat(windowWidth * modelScale, windowLength * modelScale, CV_8UC3, Scalar(fieldBGBl
    //--------------
    //--| Points |--
    //--------------
    //-- Ground Corners
    Point topLeft(0, 0);
    Point downRight(windowLength * modelScale - lineSize, windowWidth * modelScale - lineSize);
    //-- Field Corners, Middle Line and Middle Circle
    Point fieldCorner1(fieldPadding * modelScale, fieldPadding * modelScale);
    Point fieldCorner2((fieldPadding + fieldLength) * modelScale, (fieldPadding + fieldWidth) *
    Point middleLineTop((windowLength * half) * modelScale, fieldPadding * modelScale);
    Point middleLineDown((windowLength * half) * modelScale, (fieldPadding + middleLine) * mode
    Point middle((fieldPadding + fieldLength * half) * modelScale, (fieldPadding + fieldWidth *
    //-- Penalty Areas Corners
```

FieldCreate() is going to draw the line of borders and penalties and the center point of the beginning game.in order to describe function's performance, we have to set some location on the screen with points and then pass them to the opencv drawing function to coordinate the lines.

```
Mat Field::Access(){
    return Ground;
}
```

And finally there is an accessor which allows us to have access to the image matrix inside the other header files.

★ Robot

Main function of the agent that has been obliged to control the robot movement and rotation status and also check the position of the robot inside the field by considering the border.

```
//-- Robot Class
class Robot {
    private:
        double x;
        double y;
        double theta;
        int checkValue;
        string errorInfo;
        double vX;
        double vY;
        double vTheta;
        double tempX;
        double tempY;
        double tempTheta;
        double tempVX;
        double tempVY;
        double tempVTheta;
```

```
public:
    //-- Config Robot's Spawn
    void setPosition(double, d
    //-- Stores Last Position
    void storePosition(double,
    //-- Position Accessors
    double accessX();
    double accessY();
    double accessTheta();
    double accessVX();
    double accessVY();
    double accessVTheta();
    void setX(double);
    void setY(double);
    void setTheta(double);
    //-- Border Impact Check
    int borderCheck();
```

```
void Robot::setPosition(double inputX, double inputY, double inputTheta) {
    x = inputX;
    y = inputY;
    theta = inputTheta;
}
```

With this function we can set the robot position on the field also in the constructor of another header , it will be called.

```
void Robot::storePosition(double inputX, double inputY, double inputTheta,
    tempX = inputX;
    tempY = inputY;
    tempTheta = inputTheta;
    tempVX = inputVX;
    tempVY = inputVY;
    tempVTheta = inputVTheta;

}
```

storePosition() tries to save the last amounts of robots dependencies such as velocities and locations to use in other functions.

```
//-- Check Impact with Border
int Robot::borderCheck() {
    //-- if Impaction then Move to Last Position
   if (x > fieldLength / 2){
       x = tempX - 0.01;
       checkValue = -1;
   } else if (x  < -(fieldLength / 2)){
       x = tempX + 0.01;
       checkValue = 1;
   } else if (y < -(fieldWidth / 2)){
       y = tempY + 0.01;
       checkValue = 2;
   } else if (y > fieldWidth / 2){
       y = tempY - 0.01;
       checkValue = 2;
   } else {
       checkValue = 0;
   }
   switch(checkValue){
```

In this case the program checks whether the robot is inside the border or not; based on results of this checking , it will switch on some messages that will show on HelpWindow to inform us about robot status.

```
//-- Seek Movement and Rotation Changes
int Robot::state() {
    int output;
    if (tempVX != vX) {
        output = 1;
    } else if (tempVY != vY) {
        output = 2;
    }
    if (tempVTheta != vTheta){
        output = 3;
    }
    if (tempVTheta == vTheta && tempVX == vX && tempVY == vY){
        output = 0;
    }
    return output;
}
```

State function has a close connection to World.cpp in order to check if the situation of the robot is going to change or not.

```
//-- Set Robot's Velocity (Movement Velocity, Rotation Velocity) //set
void Robot::setVelocity(double inputX, double inputY, double inputTheta) {
    bool inLimit = true;
    if (inputX > maxMovementSpeed || inputY > maxMovementSpeed || inputX <
        inLimit = false;
    }
    if (inLimit == true) {
        vX = inputX;
        vY = inputY;
        vTheta = inputTheta;
    } else {
        cout << "Maximum Speed Reached !" << endl;
        errorInfo = "Maximum Speed Reached !";
    }
}
```

One of the important parts of robot header is this function !!

setVelocity receives its variable's quantities from World.cpp which the user enters with keyboard events. Although there is a absolut limitation for the range of velocities to avoid rapid movement.

```
//-- Updates Robot's Position
void Robot::update() {
    //-- Movement Part
    double globalVX = vX * cos(theta) + vY * sin(-theta);
    double globalVY = vY * cos(-theta) + vX * sin(theta);
    x = x + globalVX * refreshRate;
    y = y + globalVY * refreshRate;
    // -- Rotation Part
    // cout << theta * 180 / M_PI << endl;
    theta += vTheta * refreshRate;
    theta = modAngle(theta);
}
```

Update () function will be called in every frames of displaying the matrix ; For each axis of transition it uses a mathematical formula to change the backgroud's matrix and set the rotation value to the robot's theta. As you can see, there is a refresh rate quantity which puts a delay on robot transition and robot rotation.

```
//-- Robot X Value Implementor
void Robot::setX(double input){
    x = input;
}

//-- Robot Y Value Implementor
void Robot::setY(double input){
    y = input;
}

//-- Robot Theta Value Implementor
void Robot::setTheta(double input){
    theta = input;
}
```

End of the header , there are some implementor functions to use manually quantifying the variables inside the other headers.

★ Model

As we explained before , Model just updates with robot data; therefore most of the functions that are used in the robot file , are no longer worked in this scope. It allows us to make the code much less and short.

```
class Model {
    private:
        Mat Model;
        double x;
        double y;
        double theta;
        double vX;
        double vY;
        double vTheta;
    public:
        //-- Set Default Robot Position
        void setPosition(double, double, double);
        //-- Pass Robot's Speed to Model
        void setVelocity(double, double, double);
        //-- Update Position
        void update();
        //-- Accessors
        double accessX();
        double accessY();
        double accessTheta();
        double accessVX();
        double accessVY();
        double accessVTheta();
};
```

Implementer and accessor functions of the model are the same as the robot one but with a little difference . so robot's function's arguments are basically qualification from the keyboard and mouse events by a user , in other hand , model catch them from robot's data definitely .

★ World

The most important header file is the World, actually it contains all parameters which we should declare in the main function . at the top of this header we have to include all the other headers because they are going to be used during the process.

Also because of this type of coding (sigmention program sources) , to have an accurate connection to other classes , we have to make an object of them inside the World class private area. It means that we are declaring an object of a class as a variable of another class .

```
//-- Includes Needed Files
#include "HelpWindow.hpp"
#include "Field.hpp"
#include "Robot.hpp"
#include "Model.hpp"
```

These are necessary header files which we have to include inside it.

```
class World {
private:
    Field field;
    Robot robot;
    HelpWindow status;
    Mat realWorld;
    Planner planner;
    bool plannerMode;
public:
    World();
    int updateWindow();
    void create();
    static void mouseAttacher(int ,int ,int ,int ,void *);
    void Mouse(int ,int ,int ,int);
    Point2d pixel2Meter(Point pixel);
    Point meter2Pixel(Point2d meter);
};
```

As i mentioned above, private section consist of objects and a matrix which is going to be used as cloning matrix; to figure out what's going on exactly, you have to know that the program will draw the agent on the field; it means that a matrix will clone (copy on) on another matrix.

To generate a clear understating , you should know that Planner class depends on Robot Behavior Program !!

```
//-- Spawning Configuration in Constructor
World::World() {
    robot.setPosition(0, 0, M_PI);
    robot.setVelocity(0, 0, 0);
    field.fieldCreate();
    create();
    setMouseCallback("World", mouseAttacher, this);
    mouseFlag = 1;
    robot.storePosition(0, 0, 0, 0, 0, 0);
    // Disable planner mode
    plannerMode = false;
    updateWindow();
}
```

There is a constructor to implement default qualities and send the signal of creating the soccer field , also we will discuss what is mouseCallback.

```
//-- Updates Frames of Program
int World::updateWindow() {
    while (true) {
        //-- Update Window Frame's Refresh Rate :
        sleep_for(milliseconds((int)refreshRate * 1000));
        create();
        robot.storePosition(robot.accessX(), robot.accessY()
        if (robot.accessVX() != 0 || robot.accessVY() != 0 |
            status.updateHelpWindow(true);
        } else {
            status.updateHelpWindow(false);
        }
        status.viewPosition(robot.accessX(), robot.accessY()
        status.viewVelocity(robot.accessVX(), robot.accessVY
```

Original function of movement and rotation with all of considerations is updateWindow () , which allows us to perform transitions by its call back form of coding.

```
switch(waitKey(1)) {
    case (int('l')):
        status.resetError();
        robot.setVelocity(robot.accessVX(), robot.accessVY(),
    break;
    case (int('j')):
        status.resetError();
        robot.setVelocity(robot.accessVX(), robot.accessVY(),
    break;
```

To have a chain of frames to display , we have to create a switch case that switches on the waitKey() function of opencv.

Waitkey sets the displaying frames on a specific frame which we pass in it and in this program we force it to repeat 1 frame per second.

```
    if ( mouseFlag == 0 || mouseFlag == -1 || mouseFlag == 2) {
        robot.setVelocity(0, 0, 0);
    }
    robot.update();
} else {
    robot.resetCheck();
    status.updateError(robot.error());
}
```

End of this function parameters which are received from keyboard or mouse, pass to the robot class in order to set velocities.

```
//-- Draws the Robot on Field
void World::create() {
    //-- Creates Agent and Model
    field.Access().copyTo(realWorld);
    //-------------
    //--| World |--
    //-------------
    cv::Point agentCenterPixel = meter2Pixel(cv::Point2d(robot.accessX(), robot.accessY()));
    double agentTheta = robot.accessTheta();
    int agentDirectionXPixel;
    int agentDirectionYPixel;
    int agentRightXPixel;
    int agentRightYPixel;
    int agentLeftXPixel;
    int agentLeftYPixel;
    int robotSizePixel = robotSize * modelScale;
```

Create function works looks like the field creator function , in this case we have to set some points as agent's location and the pass them to the opencv function till draw a shape

```
  agentRightXPixel = agentCenterPixel.x - robotSizePixel *
  agentRightYPixel = agentCenterPixel.y + robotSizePixel *
  Point agentRight(agentRightXPixel, agentRightYPixel);
  //-- Point Left
  agentLeftXPixel = agentCenterPixel.x - robotSizePixel *
  agentLeftYPixel = agentCenterPixel.y + robotSizePixel *
  Point agentLeft(agentLeftXPixel, agentLeftYPixel);
  //-- DR Line
  line(realWorld, agentDirection, agentRight, Scalar(robot
  //-- DL Line
  line(realWorld, agentDirection, agentLeft, Scalar(robotB
  //-- OR Line
  line(realWorld, agentCenter, agentRight, Scalar(robotBlu
  //-- OL Line
  line(realWorld, agentCenter, agentLeft, Scalar(robotBlue
  imshow("World", realWorld);
```

```
//-- Attach Mouse to Window
void World::mouseAttacher(int event, int x, int y, int flags, void *data){
    World *pointer = reinterpret_cast<World* >(data);
    pointer -> Mouse(event, x, y, flags);
}
```

```
setMouseCallback("World", mouseAttacher, this);
```

Look at this line of code ! setMousecallback () is one of opencv's functions which receive the mouse events by user. The performance of this function is interesting ; so it will lock the program in an endless loop then wait for user to do anything with the mouse; even though it consider the mouse movement as a event parameter.

To break the loop, you can change (this) argument with (null) then it stop.

Unfortunately this is a non-class function , that means we can't create the mouse Callback as a function of a class. So what's the solution ?

In order to solve this problem , we can declare this function as static one inside that specific class and then make another function to connect with mouseCallback; inside the mouseAttacher() , we have to declare a pointer type and cast it to the specific class that contain mouseCallback; then must receive and send mouse events by this pointer to the second function.

```
//-- Offer Mouse Clicking Options
void World::Mouse(int event, int x, int y, int flags){
    if (robot.borderCheck() == 0){
        if (mouseFlag == 1){
            switch(event){
                //-- Click Left Button to Pick Agent
                case EVENT_LBUTTONDOWN:
                    mouseDistance = sqrt(pow(-robot.accessX() * modelScale
                    if (mouseDistance < clickAreaRadius) {
                        clickedColorValue = 100;
                        mouseFlag = -1;
                    } else {
                        if (plannerMode) {
                            cv::Point2d meter = pixel2Meter(cv::Point2d(x, y
                            planner.setDestination(meter);
                            planner.setState(1);
                        }
                    }
                break;
```

This is the helper function which is going to get pointer arguments …

```
else if (mouseFlag == -1) {
  switch(event){
      //-- Set Agent Position to Cursor Position
      case EVENT_MOUSEMOVE:
          robot.setX(-(x - windowLength * half * modelScale) / modelScale);
          robot.setY((y - windowWidth * half * modelScale) / modelScale);
      break;
      //-- Click Left Button to Place Agent
      case EVENT_LBUTTONUP:
          clickedColorValue = 0;
          mouseFlag = 1;
      break;
      case EVENT_MOUSEHWHEEL:
          if (getMouseWheelDelta(flags) < 0){
              robot.setTheta(robot.accessTheta() + mouseRotationValue * M_PI / 180);
          } else {
              robot.setTheta(robot.accessTheta() - mouseRotationValue * M_PI / 180);
          }
      break;
      case EVENT_MBUTTONDOWN:
          mouseFlag = 0;
      break;
  }
```

Mathematical Concept

```
double globalVX = vX * cos(theta) + vY * sin(-theta);
double globalVY = vY * cos(-theta) + vX * sin(theta);
```

This concept is totally a physics rule ; spouse that you want to move on a axis with X & Y side ; now imagine that robot transition is in fact a matrix transition , therefor moving on axis with specific angle need to consequent vector which makes from this formula :

X side = Cos(theta) * x + Sin(-theta) * y

Y side = Cos(-theta) * x + Sin(theta) * y

And at least the robot moves in the correct direction until we make it pause.

```
cv::Point2d World::pixel2Meter(cv::Point pixel) {
    cv::Point2d out;
    out.x = -(pixel.x - (windowLength * modelScale / 2)) / modelScale;
    out.y = (pixel.y - (windowWidth * modelScale / 2)) / modelScale;
    return out;
}
cv::Point World::meter2Pixel(cv::Point2d meter) {
    cv::Point out;
    out.x = (-meter.x + (windowLength * half)) * modelScale;;
    out.y = (meter.y + (windowWidth * half)) * modelScale;;
    return out;
}
```

These two functions have been obligated to transform the pixel scale to the meter scale and invert it.

Source Code

MRL Github Repository :
https://github.com/mrl-hsl/Soccer-Robot-Playground

Author Github Repository :
https://github.com/maze80/Soccer-Robot-Playground

.:: End of proposal ::.