UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CIÊNCIA DA COMPUTAÇÃO

SIMULADOR COMPUTACIONAL PARA AUXILIAR NO DESENVOLVIMENTO DE ESTRATÉGIAS DE COMPORTAMENTO PARA FUTEBOL DE ROBÔS

TRABALHO DE CONCLUSÃO DE CURSO

Fabrício Julian Carini Montenegro

Santa Maria, RS, Brasil

2015

SIMULADOR COMPUTACIONAL PARA AUXILIAR NO DESENVOLVIMENTO DE ESTRATÉGIAS DE COMPORTAMENTO PARA FUTEBOL DE ROBÔS

Fabrício Julian Carini Montenegro

Trabalho de Conclusão apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Giovani Rubert Librelotto

Santa Maria, RS, Brasil

2015

Montenegro, Fabrício Julian Carini

Simulador computacional para auxiliar no desenvolvimento de estratégias de comportamento para futebol de robôs / por Fabrício Julian Carini Montenegro. – 2015.
58 f.: il.; 30 cm.

Orientador: Giovani Rubert Librelotto
Monografia (Graduação) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Ciência da Computação, RS, 2015.

1. RoboCup. 2. Futebol de robôs. 3. Simulação Computacional. 4. Robótica. 5. Python. 6. Pygame. 7. JSON. I. Librelotto, Giovani Rubert. II. Título.

E-mail: spl.splinter@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

SIMULADOR COMPUTACIONAL PARA AUXILIAR NO DESENVOLVIMENTO DE ESTRATÉGIAS DE COMPORTAMENTO PARA FUTEBOL DE ROBÔS

elaborado por
Fabrício Julian Carini Montenegro

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Giovani Rubert Librelotto, Dr.
(Presidente/Orientador)

Cesar Tadeu Pozzer, Dr. (UFSM)

Juliana Kaizer Vizzotto, Drª. (UFSM)

Santa Maria, 13 de julho de 2015.

À minha mãe, a qual tem como melhor representação a palavra “amor”. Amor pelo filho, pelo marido, por minha irmã que mesmo não sendo fruto do seu ventre é amada como filha legítima, pelos animais de estimação, pelas plantas e flores, pela própria casa, pelos amigos, pela família e por deus, inclusive aceitando a minha descrença neste último. Obrigado, mãe, por tanto amor e apoio.

AGRADECIMENTOS

Agradeço ao professor Rodrigo da Silva Guerra através do qual pude expandir grandemente meus conhecimentos tanto nas áreas aplicadas da computação e da robótica quanto na forma de ver o mundo; ao professor Giovani Librelotto que me ajudou através de todo o processo de conclusão de curso; e aos professores da banca de avaliação que ofereceram uma grande ajuda no aperfeiçoamento do trabalho.

Agradeço à minha namorada, Adriéli Müller, que fez com que me sentisse completo e tivesse paz de espírito para realizar este projeto enfrentando os desafios que se apresentavam.

Agradeço à minha psiquiatra, Dra. Eunice Zanatta que me ajudou a voltar a ficar em pé em frente à vida numa época em que esse momento de hoje era visto como utópico e, senão impossível, ao menos improvável.

Agradeço ao meu padrasto, Marco Antônio, pelo apoio e ajuda que sempre esteve disposto a oferecer.

Agradeço, a todos os amigos cuja companhia me ajudou a seguir em frente e esquecer temporariamente os problemas quando eles afligiam minha mente.

Agradeço, por fim, à UFSM, instituição de ensino que me acolheu durante todos esses anos e que passei a considerar extensão da minha própria casa.

A todos, obrigado.

RESUMO

Trabalho de Conclusão de Curso
Ciência da Computação
Universidade Federal de Santa Maria

SIMULADOR COMPUTACIONAL PARA AUXILIAR NO DESENVOLVIMENTO DE ESTRATÉGIAS DE COMPORTAMENTO PARA FUTEBOL DE ROBÔS

AUTOR: FABRÍCIO JULIAN CARINI MONTENEGRO
ORIENTADOR: GIOVANI RUBERT LIBRELOTTO
Local da Defesa e Data: Santa Maria, 13 de julho de 2015.

A prova de futebol de robôs humanoides que acontece na RoboCup agrega conhecimento de várias áreas da tecnologia. Para o desenvolvimento de estratégias de comportamento de alto nível é possível utilizar um simulador computacional para eliminar a necessidade de se ter um robô disponível constantemente. Além de eliminar essa necessidade, o ideal seria que o simulador usado funcionasse como ambiente de desenvolvimento criando abstrações para o usuário de maneira que o código desenvolvido no simulador possa ser portado para o robô real de maneira transparente.

O problema ao necessitar tal sistema é a ausência de um simulador computacional aplicado à robótica que mantenha ao mesmo tempo um alto nível de abstração e um compromisso de compatibilidade com o hardware. Por um lado existem simuladores poderosos mas com baixo nível de abstração e por outro existem simuladores de futebol que simplificam o problema criando um bom nível de abstração mas que não estão ligados à robótica. A questão que fica é: podemos ter um simulador computacional aplicado à robótica que ao mesmo tempo possua um alto nível de abstração, simplificando a criação e testes de uma estratégia de comportamento, e mantenha uma compatibilidade com o hardware, tornando possível que o código desenvolvido no simulador seja portado para o robô de maneira transparente?

Para atingir esse objetivo, precisamos adotar um padrão de comunicação de dados entre sensores e atuadores e o agente que os possui. Este trabalho faz uso do formato *JSON*. *JSON* oferece uma representação genérica de objetos que é ao mesmo tempo simples, compacta e abrangente. Podemos simplificar as informações necessárias sobre os objetos percebidos no ambiente e sobre os comandos a serem passados aos atuadores.

Com esse objetivo em mente foi criado um simulador computacional 2D usando *python* como linguagem de programação e *pygame* como biblioteca gráfica. O nível de abstração foi atingido com sucesso fazendo possível que o programador da estratégia de comportamento mantenha o foco na lógica de seu algoritmo, abstraindo a forma como funciona o robô.

O simulador ainda provê um ambiente de criação e testes de estratégias de comportamento para o programador, mas fica pendente na verificação sobre a portabilidade do código. Apesar disso, é fácil ver que podemos adaptar o código do robô para que a comunicação com a estratégia de comportamento seja feita seguindo os padrões especificados nesse trabalho, tornando assim possível que o código criado no simulador seja portado para o robô de maneira transparente.

Palavras-chave: RoboCup. Futebol de robôs. Simulação Computacional. Robótica. Python. Pygame. JSON.

ABSTRACT

Undergraduate Final Work
Graduation Program in Informatics
Federal University of Santa Maria

COMPUTER SIMULATOR TO HELP THE DEVELOPMENT OF BEHAVIOR STRATEGIES FOR ROBOTS SOCCER

AUTHOR: FABRÍCIO JULIAN CARINI MONTENEGRO
ADVISOR: GIOVANI RUBERT LIBRELOTTO
Defense Place and Date: Santa Maria, July 13th, 2015.

The humanoid robot soccer competition that happens in RoboCup gathers knowledge from various fields of technology. For the development of high-level behavior strategies we can use a computer simulator to eliminate the need to have a constantly available robot for the work to evolve. Besides eliminating this need, the ideal scenario would be that the simulator could be used as a development environment. This could be done by creating abstractions to the user so that the code developed in the simulator can be ported to the real robot transparently.

The problem with this idea is the lack of a computer simulator applied to robotics that keeps both a high level of abstraction and a hardware compatibility commitment. On the one hand there are powerful simulators but with a low level of abstraction and on the other there are soccer simulators that simplify the problem by creating a good level of abstraction but are not related to robotics. The remaining question is: can we have a computer simulator applied to robotics that at the same time has a high level of abstraction and maintains hardware compatibility?

To achieve this goal, we need to adopt a standard data communication between sensors and actuators and the agent who owns them. This paper suggests using the JSON format. JSON provides a generic representation of objects that is at the same time simple, compact and comprehensive. We can simplify the necessary information about the objects perceived in the environment and the commands to be passed to the actuators.

With this goal in mind we created a 2D computer simulator using python as programming language and pygame as graphic library. The level of abstraction was achieved successfully making it possible for the behavior strategy programmer to keep focus on the logic of his algorithm, abstracting how the robot works.

The simulator also provides an environment for creating and testing behavior strategies, but is pending on the verification of code portability. Nevertheless, it is easy to see that we can adapt the robot code so that communication with the behavior strategy is made following the standards specified in this paper, thus making it possible for the code created in the simulator to be ported seamlessly to the robot.

Keywords: RoboCup, Robot Soccer, Computer Simulation, Robotics, Python, Pygame, JSON.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
FIFA	*Fédération Internationale de Football Association*
JSON	*JavaScript Object Notation*
MVC	*Model-View-Controller*
RCSSS	*RoboCup Soccer Simulator Server*
SDL	*Simple DIrectMedia Layer*
UDP	*User Datagram Protocol*
UFSM	Universidade Federal de Santa Maria

SUMÁRIO

1 INTRODUÇÃO

Em 2015 a Universidade Federal de Santa Maria (UFSM) irá participar pela primeira vez da *RoboCup*, evento internacional que tem como objetivo fomentar o desenvolvimento e a pesquisa nos campos da robótica e inteligência artificial, representada pela equipe *Taura Bots*. Para tal, foi desenvolvido um robô humanoide que participará da prova de futebol de robôs.

O desenvolvimento de um robô agrega conhecimentos de diversas áreas da tecnologia entre engenharia elétrica, mecânica, de controle e automação e computação e, uma vez que o projeto envolve tantas áreas, a equipe acaba subdividindo-se em equipes menores.

Muitas das subequipes precisam do uso constante do robô para que o trabalho evolua. Um exemplo disso é a equipe que programa o controle de caminhada. É possível alterar parâmetros do algoritmo de caminhada e teorizar sobre o resultado na prática, mas esse resultado só será conhecido quando for feito um teste no robô. Isso faz com que o robô seja um recurso escasso. Tratando-se da equipe de desenvolvimento da estratégia de comportamento do robô, formada por alunos de ciência da computação, esse problema pode ser solucionado com o uso de um simulador computacional.

Surge então a ideia de utilizar um simulador computacional para o desenvolvimento de estratégias de comportamento de alto nível, eliminando a dependência do robô como recurso. Isso acaba levando a outro problema que é a falta de um simulador computacional aplicado à robótica que ofereça o nível de abstração desejado.

Durante a pesquisa foram encontrados alguns simuladores que foram candidatos a uso pela equipe. Um deles é o Gazebo, um simulador 3D extremamente poderoso capaz de simular dinâmica e física do mundo real. O problema do Gazebo, quando aplicado à situação descrita anteriormente, é que ele possui um baixíssimo nível de abstração. Um código testado no robô simulado dentro do Gazebo certamente funcionaria em uma versão real do robô, o problema é que precisamos criar dois robôs, um real e um simulado. Portanto, apesar de útil, o Gazebo não simplifica a tarefa de criar uma estratégia de comportamento.

Em contraste ao Gazebo, foi encontrado o *RoboCup Soccer Simulator Server* (RCSSS) que possui um alto nível de abstração. Ele é usado pela própria *Robocup* na categoria de simulação de futebol e, à primeira vista parece ser exatamente o que é preciso para testar estratégias de comportamento. O problema do RCSSS é que ele não é um simulador de robótica, mas sim um simulador de futebol. São dois problemas distintos e, por esse motivo, o código que des-

creve uma estratégia de comportamento no simulador não necessariamente irá funcionar quando aplicado a um robô real.

Além disso, o ideal seria ter-se um simulador que funcione como ambiente para desenvolvimento de estratégias de comportamento de alto nível de abstração, oferecendo ao programador uma *API* de maneira que o código desenvolvido no simulador seja usado diretamente no robô.

O que torna perceptível real problema: é possível ter-se um simulador computacional aplicado à robótica que ofereça ao programador um nível alto de abstração mas que mantenha uma compatibilidade com o hardware? Este problema serve então como motivação para o trabalho aqui apresentado.

Durante a execução do trabalho foi desenvolvido um sistema que tem como objetivo fornecer ao programador um ambiente de simulação de maneira que ele não fique diretamente dependente do robô para a execução de testes. O objetivo ainda vai além, pois criando uma abstração sobre a forma como o robô funciona facilitamos ainda mais o desenvolvimento de uma estratégia de comportamento. O programador abstrai a forma como o robô coleta dados sobre o ambiente e como a programação dos movimentos do robô funciona, por exemplo, podendo assim manter o foco apenas na lógica de sua estratégia. Não importa se o programador recebe dados de um simulador ou de um robô real e não importa se ele envia comandos a um robô simulado ou a um robô real, a lógica da sua estratégia é a mesma em ambas as situações então o código que ele precisa escrever também deveria poder ser o mesmo.

A comunicação de dados, apresentada no texto que se segue, foi feita usando o formato JSON e é o ponto central da abordagem aqui apresentada. Através do uso de uma comunicação de dados que segue um formato padrão, simples e compacto, é possível ao mesmo tempo criar um alto nível de abstração para o usuário que irá simular sua estratégia e manter uma camada de compatibilidade com o sistema real. Para o teste dessa abordagem foi criado um sistema em Python. O código descrito à frente serve meramente como meio para o teste da abordagem apresentada e vários simuladores poderiam ser criados usando várias linguagens de programação diferentes. A ideia central é a abstração de partes do sistema através do uso de um formato padronizado de comunicação de dados.

2 TÓPICOS DE INTERESSE

2.1 *RoboCup*

A *RoboCup*, chamada no Brasil de Copa do Mundo de Robôs, tem como objetivo fomentar o desenvolvimento e a pesquisa nas áreas de robótica e inteligência artificial (KITANO et al., 1998). A competição oferece várias categorias envolvendo robôs para resgate, robôs no ambiente de trabalho e robôs em ambiente doméstico, mas a liga que mais atrai o grande público é o futebol de robôs. Dentro da liga de futebol de robôs existem ainda várias categorias, como simulação de futebol, futebol de robôs com rodas e futebol de robôs humanoides, de vários tamanhos.

Em 1997, um evento colocava frente à frente pela segunda vez Garry Kasparov, considerado o melhor jogador de xadrez de todos os tempos, e *Deep Blue*, um computador desenvolvido pela *IBM* para jogar xadrez (CAMPBELL; HOANE; HSU, 2002). No primeiro confronto a máquina venceu apenas uma das partidas disputadas, mas desta vez ela saiu vencedora absoluta do confronto e, ao transpor esse desafio, a pesquisa em inteligência artificial procurava um novo objetivo que fosse desafiador o suficiente para expandir o estado da arte. Foi então que a *RoboCup* surgiu propondo o objetivo de que à metade do século XXI uma equipe de robôs humanoides seja capaz de vencer uma partida de futebol contra a equipe (de humanos) campeã mundial do mesmo ano seguindo as regras oficiais da FIFA (KITANO et al., 1998). Atualmente este objetivo não parece tão próximo de ser alcançado, mas a cada ano a comissão organizadora altera algumas das regras da competição afim de aumentar o desafio, fazendo com que as equipes tenham que transpor barreiras cada vez mais altas.

Em 2015 a *RoboCup* será sediada em Hefei, na China e contará com a participação da UFSM representada pela equipe *Taura Bots*. A equipe competirá em parceria com a equipe alemã *WF-Wolves* da Universidade de Ostfalia. A participação da UFSM será na liga de futebol de robôs humanoides na categoria *Kid-Size* que contempla robôs de 40 a 90 centímetros de altura.

2.2 Robótica e Inteligência Artificial

Robótica e inteligência artificial são dois assuntos ligados intimamente. A teoria de *embodiment*, aqui traduzido livremente como *personificação* no sentido de possuir um corpo,

sugere que agentes autônomos do mundo natural são em sua maioria seres biológicos e que sua inteligência está diretamente ligada a seus corpos e a forma com a qual eles interagem com o ambiente. Dessa forma, para criar agentes com inteligência artificial, precisamos dar um corpo artificial, ou robótico, a esses agentes. A personificação implica que o agente está continuamente sujeito a quaisquer influências do ambiente e podemos estudar, por exemplo, como um organismo adquire experiência: conhecimento sobre o ambiente obtido ao interagir com ele (PFEIFER; SCHEIER; ILLUSTRATOR-FOLLATH, 2001).

Um agente é qualquer coisa que pode ser vista como aquela que percebe o ambiente através de sensores e age nesse ambiente através de atuadores. Um agente humano tem olhos, ouvidos e outros órgãos funcionando como sensores; e mãos, pernas, boca e outras partes do corpo como atuadores. Um agente robótico usa câmeras como sensores e motores variados como atuadores. Um *software* agente tem palavras codificadas em bits como suas percepções e suas ações (RUSSELL; NORVIG; INTELLIGENCE, 1995).

Na Figura 2.1 estão descritos ambiente e agente. O agente interage com o ambiente através de sensores e atuadores.

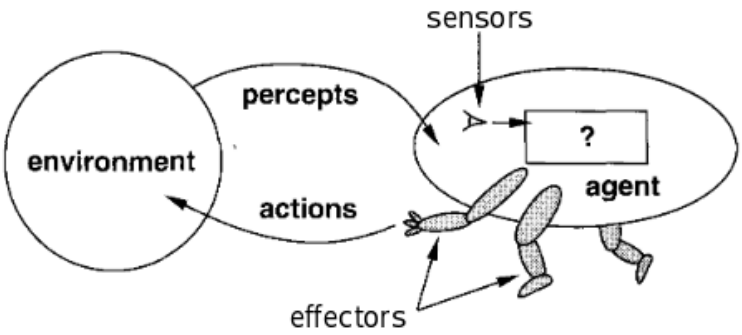

Figura 2.1: Um agente interage com o ambiente através de sensores e atuadores.

Para que seja criado um simulador computacional com alto nível de abstração focado somente na definição do comportamento, é preciso simular o ambiente, os sensores e os atuadores, deixando o usuário responsável apenas pela criação do agente. No caso do nível de abstração que desejamos nesse trabalho, o agente é unicamente responsável pela tomada de decisões. Dessa maneira, o corpo, formado por sensores e atuadores, não faz parte do agente pois ele representa apenas a mente do robô. Nessa abordagem os sensores e atuadores fazem parte do ambiente.

2.3 Simulação Computacional

Simulação é a imitação de uma operação de um processo ou sistema do mundo real através do tempo. Uma simulação computacional é uma tentativa de modelar uma situação hipotética ou real em um computador de tal maneira que ela possa ser estudada para averiguar como o sistema funciona. Mudando variáveis da simulação, predições podem ser feitas sobre o comportamento do sistema. Ele é uma ferramenta para investigar virtualmente o comportamento do sistema em estudo (BANKS; CARSON; NELSON, 2000).

Simuladores têm representado um papel essencial na pesquisa em robótica como ferramentas de testes eficientes e eficazes de novos conceitos, estratégias e algoritmos (KOENIG; HOWARD, 2004). Tratando-se do desenvolvimento do robô que irá competir na China, alguns simuladores foram estudados e o primeiro a ser citado é o Gazebo.

O simulador Gazebo é projetado para preencher o nicho de simuladores que são altamente capazes e facilmente adaptáveis criando um ambiente multi-robô 3D dinâmico capaz de recriar mundos complexos (KOENIG; HOWARD, 2013). A complexidade deste simulador é extremamente útil durante a fase de projeto do robô pois ele permite simular componentes como circuitos e partes mecânicas do robô.

Outro simulador a ser citado é o *Soccer Server* (ITSUKI, 1995). O *Soccer Server* é desenvolvido pela própria *RoboCup Organization* e usado na liga de simulação. Ele fornece um ambiente onde dois times de jogadores se confrontam sendo controlados por vários tipos de sistemas. Cada sistema conecta-se ao servidor como um cliente via rede de computadores. O campo de futebol e todos os objetos simulados são bidimensionais (ITSUKI, 1995). Apesar de ser um bom passo inicial para o desenvolvimento de uma inteligência artificial que aborde o problema de futebol de robôs, o *Soccer Server* acaba provendo uma plataforma muito genérica para a criação de uma estratégia de comportamento de um robô real, pois desconsidera algumas características do robô, como a velocidade com a qual ele se move e com a qual ele gira em seu próprio eixo para mudar de direção. Além disso ele não leva em consideração a capacidade que o robô da UFSM tem de andar e girar ao mesmo tempo.

Pelos motivos citados, nenhum dos dois simuladores encaixa-se nas necessidades da equipe *Taura Bots*.

2.4 *Python* e *Pygame*

Para este projeto foi adotado *Python* como linguagem de programação pela sua simplicidade, suave curva de aprendizado e alto nível de abstração. *Python* foi projetada em 1990 por Guido van Rossum e é uma linguagem de programação orientada a objeto, interativa e interpretada. Ela fornece estruturas de dados de alto nível como listas e *arrays* associativos, tipagem dinâmica e associatividade dinâmica, módulos, classes, exceções, gerenciamento automático de memória, entre outras funcionalidades. Possui uma sintaxe simples e elegante e ainda assim é considerada uma linguagem de programação poderosa e de propósito geral (SANNER et al., 1999).

Para a interface gráfica foi escolhido trabalhar com a biblioteca *Pygame*. *Pygame* é um conjunto de módulos *Python* altamente portável projetado para a criação de jogos. *Pygame* adiciona funcionalidades em uma camada acima da biblioteca *SDL*, o que permite que sejam criados jogos e programas multimídia utilizando *Python* como linguagem de programação (SHINNERS, 2011).

2.5 *JSON*

JSON (*JavaScript Object Notation*) é um formato de texto para serialização de dados estruturados. Ele é derivado de um objeto de literais de *JavaScript*, conforme definido no *ECMAScript Programming Language Standard, Thrid Edition*.

JSON pode representar quatro tipos primitivos: *strings*, números, variáveis booleanas e *null*; e dois tipos estruturados: objetos e *arrays*. Uma *string* é uma sequência de zero ou mais caracteres *Unicode*. Um objeto é uma coleção desordenada de zero ou mais pares nome-valor, onde um nome é uma *string* e um valor é uma *string*, número, variável booleana, *null*, objeto, ou *array*.

O termo "objeto"e "*array*"vêm de convensões de *JavaScript*. O design do *JSON* tinha como objetivo que ele fosse mínimo, portável, textual e um subconjunto de *JavaScript* (CROCKFORD, 2013).

2.6 Sumário do Capítulo

A *RoboCup* oferece um grande desafio para desenvolvedores e cientistas do mundo inteiro expandindo o estado da arte nas áreas de robótica e inteligência artificial. Para agilizar e facilitar o desenvolvimento de uma estratégia de comportamento de alto nível, a equipe da UFSM que participará da competição em 2015, fez uso de um simulador computacional.

O simulador é responsável pelo ambiente e pelos sensores e atuadores do robô simulado, deixando o programador responsável apenas pela programação do agente que controla o robô simulado.

O simulador foi feito utilizando como linguagem de programação *Python* e como biblioteca gráfica *Pygame*. A comunicação de dados foi feita utilizando o formato *JSON*, que além de compacto e portável, gera uma abstração conveniente para o programador da estratégia. As ferramentas utilizadas neste trabalho foram escolhidas durante uma fase de levantamento de requisitos e de planejamento. Essas fases são descritas a seguir, no capítulo de metodologia.

3 METODOLOGIA

O simulador se propõe a oferecer um ambiente de simulação para que o usuário crie uma estratégia de comportamento para um robô que joga futebol. Diferentemente dos simuladores citados anteriormente o simulador aqui proposto busca atingir um alto nível de abstração e, ao mesmo tempo, manter a compatibilidade com o hardware.

Para atingir este objetivo, podemos criar um simulador que interprete comandos e que entregue dados ao usuário da mesma forma que o robô faz. Para essa abordagem, precisamos saber como o comportamento de um robô que joga futebol é feito para podermos imitar o comportamento de sensores e atuadores de maneira transparente.

Quanto aos sensores, levamos em consideração apenas a câmera, que é essencial para que o robô perceba o ambiente. Para que a simulação gere uma abstração para o usuário, é necessário simular todo o processo feito desde a captura de uma imagem até a detecção de objetos nela. Para o programador do comportamento, é necessário, por exemplo, saber a posição da bola, mas ele não precisa saber como essa bola é detectada no ambiente. Para que essa informação chegue ao controle do comportamento, primeiramente a câmera captura uma imagem do ambiente. Essa imagem então é tratada através do uso de filtros que facilitam o uso de algoritmos de detecção de objetos. Quando temos a imagem tratada, o algoritmo de detecção de objetos nos entrega dados relativos aos objetos detectados. São esses dados que precisam ser conhecidos pelo usuário, abstraindo a forma como a coleta deles foi feita. O programador precisa saber sobre os objetos apenas qual o tipo de objeto visto e a posição desse objeto em relação ao robô.

Tratando-se dos atuadores, o robô é capaz de mover braços, pernas e pescoço, mas quais desses movimentos são realmente importantes para o desenvolvedor do comportamento? A posição em que a cabeça se encontra é uma informação valiosa pois alterando essa posição podemos buscar objetos no ambiente, então o programador deve ter acesso a essa posição. O funcionamento dos braços são irrelevantes para o comportamento pois, com o foco em jogar futebol, eles influenciam apenas no equilíbrio e, caso o robô sofra uma queda, os braços o ajudam a erguer-se. O equilíbrio não é relevante para o programador pois o controle de caminhada é uma parte muito complexa da programação do robô e gerando uma abstração sobre essa parte, facilitamos o desenvolvimento da estratégia de comportamento.

Já que não levamos em conta detalhes sobre a caminhada, o funcionamento das pernas

também é mantido em uma camada inacessível pelo programador. Entretanto, a principal interação entre o jogador e o ambiente em uma partida de futebol é o chute na bola. Por causa disso, precisamos ter uma maneira de indicar quando o robô deve dar um chute.

A informação sobre a caminhada à qual o usuário tem acesso fica então limitada à direção e velocidade de movimento do robô. Além disso um robô humanoide pode girar em seu próprio eixo sem sair do lugar ou enquanto anda em uma direção e essa capacidade pode ser útil ao programador, portanto ele precisa ter acesso a essa informação também.

Sendo assim, aplicando uma abstração sobre o funcionamento do robô, o programador tem acesso às seguintes informações: posição e tipo de objetos detectados no ambiente, direção e velocidade de caminhada além de um ângulo para rotação do corpo, rotação da cabeça em relação ao corpo e, por fim, controle do chute.

Para fazer com que o usuário possa usar o mesmo código para controlar um robô simulado e o robô real, temos que fazer com que tanto a coleta de dados sobre o ambiente feita pelo agente quanto o envio de comandos do agente para o corpo do robô sejam feitos de maneira transparente. Para tal, temos que encontrar um formato padrão de comunicação de dados. É para este fim que foi usado o formado JSON. A definição do JSON oferece uma maneira simples e compacta de transmitir informações possibilitando uma simplificação na comunicação de dados. O robô real pode ser programado de tal maneira que o controle de caminhada possa receber como argumento um objeto JSON indicando apenas a direção e velocidade de caminhada. A visão do robô também pode ser programada de maneira que ela detecte objetos e os comunique ao agente através de objetos JSON. Se a comunicação no robô real seguir esse padrão, podemos facilmente utilizar o mesmo formato de comunicação no simulador deixando a programação da estratégia transparente.

Para desenvolver um sistema de simulação que utilize a abordagem descrita até aqui, foram coletados os requisitos do sistema informados na seção seguinte.

3.1 Requisitos

O ponto inicial do planejamento do simulador foi uma interface gráfica 2D simulando o campo de futebol onde o usuário pode adicionar objetos afim de criar um cenário para testes de uma certa estratégia de comportamento. A simulação pode ser feita em 2D pois os robôs humanoides de hoje em dia têm dificuldade em fazer chutes nos quais a bola se eleve no ar deixando o chão, por isso a altura dos objetos não é relevante e podemos abstrair essa informação.

Os objetos passíveis de serem criados nesse campo simulado são objetos presentes no ambiente de uma partida de futebol de robôs e inicialmente foram listados em quatro categorias: bola, robô, poste e objeto não identificado. A lista pode ser aumentada com o passar do tempo e o aperfeiçoamento do simulador já que a equipe pretende continuar usando o sistema nas próximas edições da competição. Durante o desenvolvimento do sistema de visão a equipe identificou, por exemplo, a necessidade de representar vértices de interesse entre as linhas do campo. Foi decidido, entretanto, que para o estágio inicial de desenvolvimento no qual a inteligência artificial se encontra, as quatro categorias de objetos seriam suficientes para a tomada de decisões.

Mais uma necessidade identificada pela equipe de visão é de que os objetos simulados possuam um raio de incerteza controlado pelo usuário pois muitas vezes algoritmos de visão para detecção de objetos não retornam informações precisas. Quando a posição do objeto percebido pelo robô for entregue ao agente que o controla, essa posição deve possuir um ruído dentro do raio de incerteza para simular situações onde esse problema ocorre e analisar como o agente reage a isso.

Uma vez que o cenário estiver montado, um processo que representa um agente deve poder conectar-se à simulação do mundo ligando-se a um robô presente no cenário. Após conectado, o agente deve receber dados sobre o ambiente percebido pelo robô através de uma camada de abstração que faz a comunicação. Vários agentes podem conectar-se à simulação do mundo ao mesmo tempo, fornecendo uma característica de servidor, semelhante ao funcionamento do *RCSSS*, citado anteriormente.

O simulador pode, então, ser dividido em duas partes: uma parte onde o usuário pode criar um cenário onde deseje testar sua estratégia de comportamento e que simula os sensores do robô; e outra parte que percebe o mundo através das informações recebidas dos sensores e que funciona como uma camada de abstração apenas entregando ao agente os dados que sejam relevantes para o comportamento. O caminho inverso também é verdadeiro, quando o agente toma uma decisão ela deve ser transmitida pela camada de abstração que simula o corpo do robô e então o comando é entregue ao ambiente para que ele possa reproduzir a ação executada pelo robô.

A Figura 3.1 representa a arquitetura do sistema, e é possível visualizar como a comunicação funciona entre o ambiente e uma camada de abstração que irá entregar os dados ao agente desenvolvido pelo usuário.

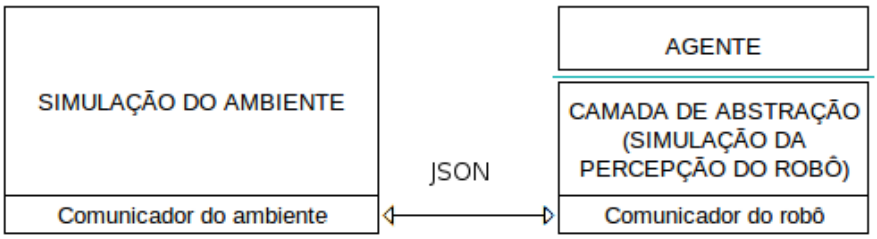

Figura 3.1: Arquitetura do sistema.

É ainda um requisito que o sistema faça a comunicação entre o ambiente simulado e a simulação do robô através de um protocolo *UDP* simplificando a comunicação de dados e dando um aspecto de aplicação de tempo real, já que dados transmitidos via protocolo *UDP* não precisam de confirmação de entrega. A comunicação via rede ainda cria a possibilidade de que o robô real seja testado no ambiente simulado porque enquanto um módulo de comunicação roda no robô, o módulo de simulação do ambiente pode rodar em outra máquina.

Para a representação de objetos no ambiente simulado podemos considerar as posições dos objetos como pontos representados em coordenadas cartesianas em relação a uma origem no espaço 2D. A origem, para este simulador, foi adotada como sendo a origem da tela, no vértice do topo esquerdo.

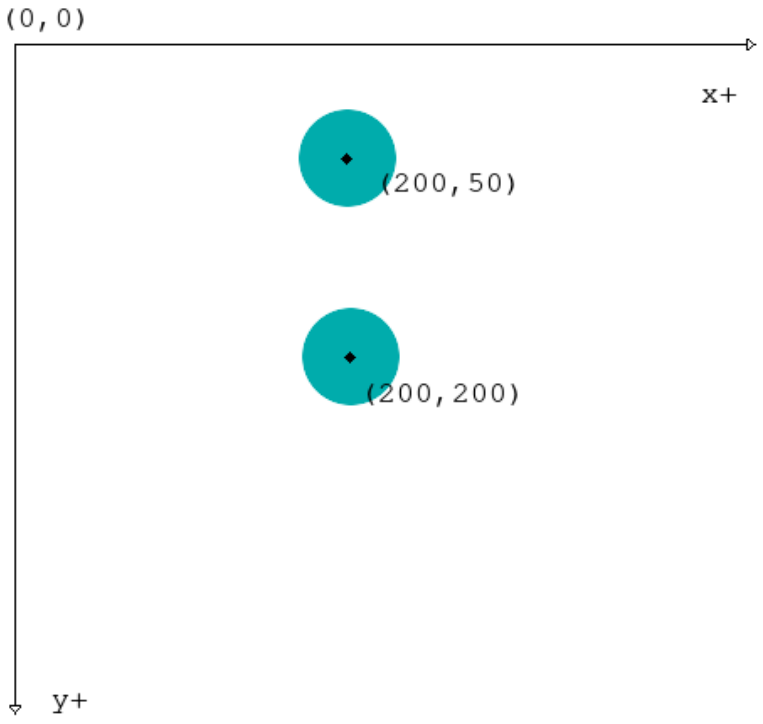

Figura 3.2: Objetos representados em uma perspectiva global no espaço 2D.

Quando as informações sobre os objetos presentes no mundo forem entregues ao agente,

este deve receber as posições dos objetos em uma perspectiva egocêntrica. Para exemplificar esse tipo de perspectiva egocêntrica, quando movemos nossos olhos para a direita, os objetos estáticos presentes no ambiente são percebidos cada vez mais à esquerda do nosso campo de visão. Se levarmos em consideração a posição dos objetos em nosso campo de visão ao invés de considerar suas posições em relação a um ponto de origem no ambiente, então quem se move são os objetos e não nossos olhos.

Se considerarmos o mundo como um plano bidimensional, podemos representar as posições dos objetos em coordenadas polares, o que facilita o modo como lidamos com essa informação. Podemos considerar o ângulo zero como a frente do agente e considerar que o ângulo aumenta em sentido anti-horário, ou seja, para a esquerda.

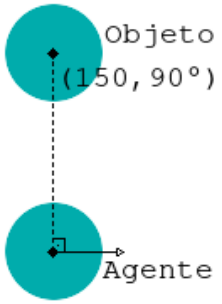

Figura 3.3: Posição do objeto em uma perspectiva egocêntrica do agente voltado para a direita.

Considerando o centro do campo de visão do agente como ângulo zero, se fizermos um corte na linha deste ângulo zero, objetos à esquerda no campo de visão terão como segunda coordenada um ângulo positivo entre 0 e π e objetos à direita um ângulo negativo entre 0 e $-\pi$.

Para facilitar a representação e o manuseio de pontos no espaço 2D do ambiente e a conversão desses pontos para um espaço egocêntrico de coordenadas polares, foi criada uma classe de vetores 2D. A classe *Vector2* armazena a representação do ponto em coordenadas retangulares e polares ao mesmo tempo, recalculando a posição em uma das duas representações quando a outra tiver uma coordenada alterada. Se, por exemplo, for alterada a coordenada x de um vetor, sua representação em coordenadas polares será recalculada de forma que quando os valores de r e θ forem acessados estes estejam corretos. A classe ainda implementa soma de vetores e multiplicação por escalares e está documentada em anexo.

3.2 Sumário do Capítulo

O simulador procura tornar transparente a criação e os testes de estratégias de comportamento para futebol de robôs, fazendo com que seja possível que um código a ser testado em um simulador seja usado diretamente no robô real. A comunicação de dados no formato JSON é essencial para que esse resultado seja alcançado quando seguimos a abordagem descrita até aqui.

O sistema deve simular um ambiente bidimensional onde objetos são adicionados afim de criar um cenário para testes de uma estratégia de comportamento. Quem controla o comportamento de um robô simulado no ambiente é um agente. O agente conecta-se ao ambiente através de um outro processo que simula o corpo do robô e a forma como ele comunica-se com o ambiente.

Dados os objetivos do sistema e a forma como ele deve funcionar, será abordada sua modelagem.

4 MODELAGEM: SIMULAÇÃO DO AMBIENTE

O simulador possui uma arquitetura *MVC* (Modelo-Visão-Controlador) de forma que cada parte do sistema fique separada das outras melhorando a organização do código. Nas classes referentes aos modelos ficam as definições dos objetos a serem instanciados no mundo; nas classes referentes à visão ficam definidos os métodos que geram a parte gráfica do sistema; e nas classes referentes aos controladores ficam a lógica que une modelos e visão e o comportamento do sistema em resposta a interações do usuário.

Considerando a divisão do sistema em uma parte que simula o ambiente e outra parte que simula o corpo do robô controlado pelo agente a ser testado, será feita, primeiramente, uma explicação sobre a simulação do ambiente e, posteriormente, uma explicação sobre a simulação do corpo do robô. A Figura 4.1 mostra o diagrama de classes da parte do sistema que gerencia a simulação do ambiente.

A simulação do ambiente é chamada de *World* e se dá, respeitando a arquitetura *MVC*, em quatro módulos *python*: *Model, View, Controller* e *Communicator*. Ao iniciar a aplicação o controlador cria uma instância do modelo e da visão e, toda vez que um robô for adicionado ao ambiente, um comunicador é criado para enviar dados para o agente que irá controlar aquele robô.

A seguir uma explicação sobre cada um dos módulos descritos no diagrama de classes de *World* além de uma explicação sobre as classes contidas nesses módulos e sobre como elas interagem entre si.

4.1 *Model*

Conforme ilustra a Figura 4.1, o módulo *Model* define três classes que juntas definem como um objeto é representado, como um robô é representado e como objetos e robôs presentes no ambiente simulado são armazenados.

4.1.1 A classe *W_BaseObjectModel*

W_BaseObjectModel define o que é considerado um objeto a ser simulado no mundo. Um objeto tem uma posição (*position*) no espaço 2D da tela, representada em coordenadas retangulares; um tipo (*kind*) relativo à classe de objetos à qual esse objeto pertence sendo elas:

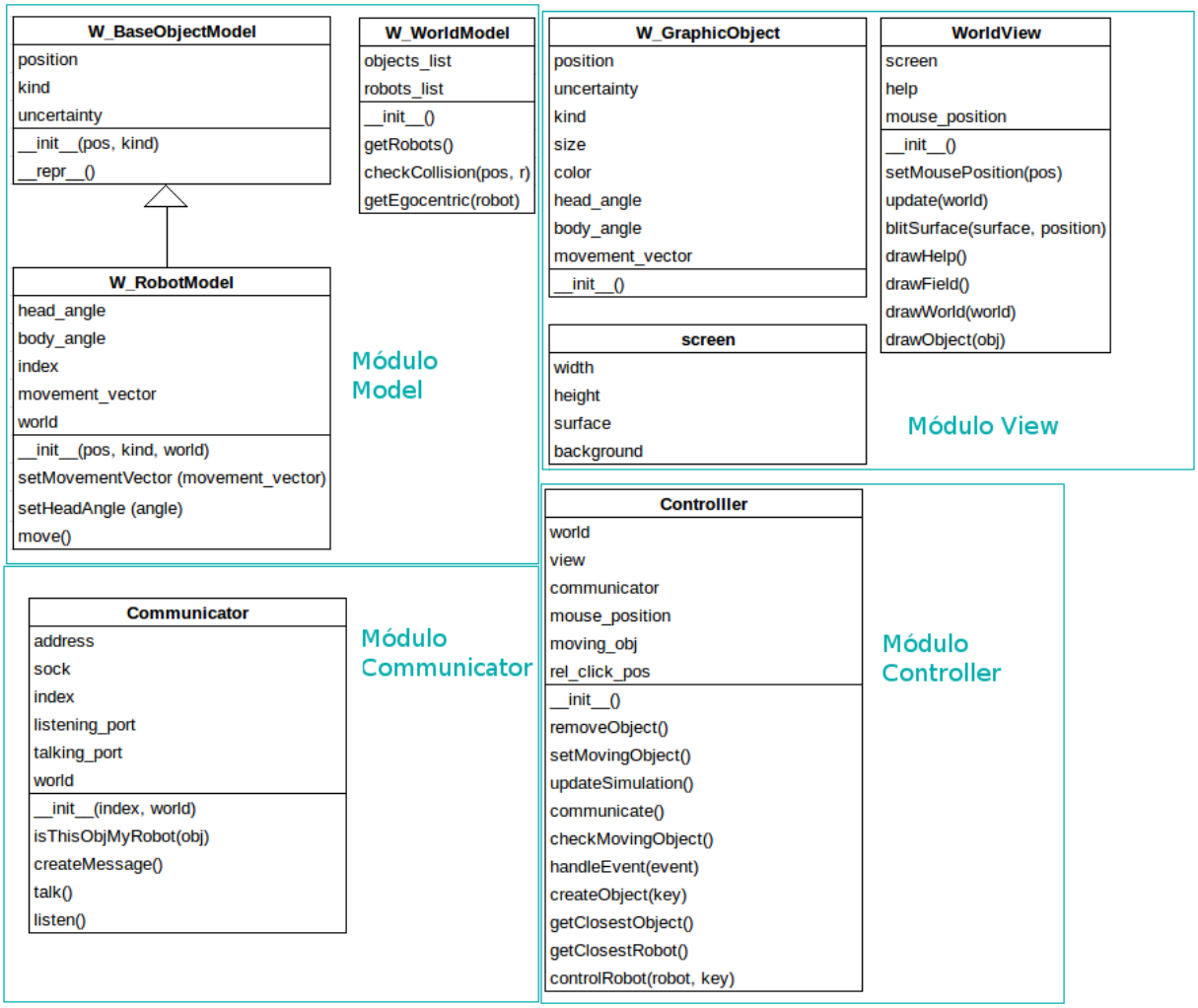

Figura 4.1: Módulos de *World*.

bola, poste, robô e objeto não identificado; e um raio de incerteza (*uncertainty*).

Em *python*, o construtor da classe é identificado pela *string* reservada *__init__* e o método que contém a representação da classe em forma de *string* identificado por *__repr__*, portanto a explicação desses métodos será omitida do texto para fins de simplicidade.

4.1.2 A classe *W_RobotModel*

A classe *W_RobotModel* define o que é um robô, que não deixa de ser um objeto e, por isso, herda atributos de *W_BaseObjectModel*. O ângulo do corpo (*body_angle*) do robô fornece a informação sobre a frente do robô e representa a rotação que ele deve sofrer considerando que o ângulo zero é quando o objeto está voltado para a direita se observado na perspectiva global. Uma bola, por exemplo, não tem uma frente e por isso a definição básica de objeto não possui esse atributo. Pelo mesmo motivo apenas o robô possui o atributo *head_angle*.

Um robô possui ainda um atributo identificador (*index*) cujo uso será explicado mais à frente neste texto. O atributo *world* guarda uma referência à instância da classe *W_WorldModel*, que será explicada em seguida, onde estão as informações sobre o ambiente. A classe que define robô também possui um *movement_vector* que representa a intenção de movimento do robô. Essa intenção é setada pelo agente que estará controlando aquele robô e sua posição no ambiente é atualizada ao ser chamado o método *move*, que usa o vetor de movimento como incremento para a posição atual do robô.

4.1.3 A classe *W_WorldModel*

W_WorldModel define as informações a serem armazenadas sobre o ambiente. Uma lista armazena os objetos presentes no ambiente. O método *getRobots* retorna a lista de robôs presentes no ambiente. O método *checkCollision* é usado para testar a colisão de objetos uns com os outros e com o ponteiro do mouse. O método *getEgocentric* recebe um robô e retorna os objetos do ambiente com as posições convertidas para a perspectiva egocêntrica desde robô.

Sabendo como os objetos são definidos e como eles são armazenados no ambiente, será explicada sua representação gráfica.

4.2 *View*

O módulo de visão implementa e gerencia características referentes à interface gráfica da aplicação. Separando as informações sobre os objetos utilizadas no modelo e no controlador das informações sobre a representação gráfica destes objetos utilizadas na visão, damos ao sistema uma modularidade maior e uma melhor organização.

A configuração da tela a ser exibida não é relevante para a forma como a lógica do sistema lida com os objetos e não misturar as duas partes ajuda na compreensão do código.

4.2.1 A classe *W_GraphicObject*

Para a separação entre a parte lógica e a parte gráfica, existe a classe *W_GraphicObject* que armazena informações sobre o objeto a ser desenhado na tela. Por esse motivo os atributos desta classe são muito parecidos com os atributos das classes *W_BaseObjectModel* e *W_RobotModel*. Devido à flexibilidade do *python*, os atributos referentes ao objeto caso ele seja um robô podem ser alocados somente se o objeto a ser desenhado realmente é um robô,

caso contrário, eles não ocuparão memória.

Os únicos atributos que dizem respeito somente à parte gráfica são *size*, que indica o tamanho do objeto a ser desenhado na tela de acordo com o tipo do objeto, e *color*, que indica a cor do objeto, também de acordo com o tipo do objeto.

Todos os objetos são representados como círculos.

4.2.2 A classe *WorldView* e o dicionário *screen*

Conforme ilustra a Figura 4.1, a classe *WorldView* possui um atributo chamado *screen* que armazena informações sobre a tela na forma de um dicionário de dados. As informações sobre a tela são: largura, altura, cor de fundo e uma instancia de *surface* que é a forma com a qual o *pygame* representa superfícies gráficas.

O atributo *help* é uma variável booleana que indica se o texto de ajuda ao usuário deve ser exibido na tela e o atributo *mouse_position* armazena a posição do ponteiro do mouse em relação à tela, sendo que essa posição é atualizada através do método *setMousePosition*, que é chamado sempre que ocorrer um evento do tipo *mouse_move* definido pelo *pygame*.

O método *update* é chamado a cada *frame* e redesenha a tela usando os demais métodos da classe. *blitSurface* é a forma como o *pygame* "cola"superfícies na tela formando camadas; *drawHelp* exibe o texto de ajuda na tela se o atributo *help* for verdadeiro; *drawField* preenche a tela com a cor de fundo e adiciona marcadores que indicam a posição do cursor, para a conveniência do usuário; *drawWorld* faz uma iteração por todos os objetos contidos no atributo *world* que *update* recebe do controlador e chama *drawObject*, que desenha o objeto na tela de acordo com o tipo de objeto que ele é, para cada objeto listado no mundo.

4.2.3 Demonstração da tela do programa

Na situação demonstrada a seguir, estão presentes no ambiente um robô, uma bola, um poste e um objeto não identificado. O robô é representado por um círculo parte azul e parte preto. A parte preta do robô representa suas costas, e a parte azul indica a frente do corpo do robô. É possível ver que na situação demonstrada o robô está com a cabeça, representada pela reta vermelha, voltada para sua esquerda, e como o campo de visão, representado pelas retas cinzas, é afetado.

A bola é representada em branco, enquanto o poste é representado em laranja e o objeto não identificado em preto. É possível visualizar que os objetos possuem uma circunferência

semi-transparente ao seu redor que representa o nível de incerteza. A forma como essa incerteza pode ser interpretada é que, ao receber a informação sobre o ambiente, o agente perceberá a posição do objeto em algum lugar aleatório dentro do raio de incerteza.

Na situação é possível ver o texto de ajuda ao usuário e ainda os marcadores de posição do cursor representados como pequenos triângulos nas arestas inferior e esquerda da tela.

Figura 4.2: Demonstração da tela do programa.

4.3 *Communicator*

A classe *Communicator* controla a comunicação entre o ambiente simulado e os agentes conectados a esse ambiente.

Uma instância desta classe é criada para cada robô presente no ambiente e fica ouvindo dados em uma porta de rede definida. Quando a instância recebe algum dado naquela porta, ela interpreta que um agente está pedindo dados sobre o robô ao qual este comunicador pertence. É uma mente tentando conectar-se a um corpo.

Para a comunicação via rede, a classe *Communicator* possui um endereço, uma instância da classe *socket* do *python*, o índice do robô o qual ela representa, as portas para receber e enviar dados e, por fim, uma referência à instância de mundo. O endereço, por padrão, é definido como o *localhost* para o funcionamento em uma única máquina, mas pode ser redefinido pelo usuário caso ele queira simular o sistema em uma máquina e conectar seu agente a partir de outra máquina. As portas usadas para a comunicação entre o robô e seu agente seguem a seguinte fórmula:

$$listening\_port = INITIAL\_PORT + (index * 2) \tag{4.1}$$

$$talking\_port = listening\_port + 1 \tag{4.2}$$

Como cada robô ocupará duas portas para comunicar-se com seu agente, sendo uma para receber dados e outra para enviar dados, é utilizado o dobro do índice do robô.

Por exemplo, se a porta inicial usada for a porta 20000, o robô com índice 0 terá *listening_port = 20000* e *talking_port = 20001*. O robô de índice 1 terá *listening_port = 20002* e *talking_port = 20003* e o robô de índice 10 terá *listening_port = 20020* e *talking_port = 20021*.

Quanto aos métodos da classe, *isThisObjMyRobot* é um método de conveniência para comparações. *createMessage* retorna a mensagem a ser enviada pela rede contendo a lista de objetos e suas posições já convertidas para a visão egocêntrica do robô além do ângulo atual da sua cabeça. Os métodos *talk* e *listen* implementam o envio e o recebimento de dados, respectivamente.

4.4 Comunicação no formato *JSON*

Uma característica importante do sistema é ó uso do formato *JSON* para a comunicação de dados. O formato *JSON* representa objetos em uma *string*, o que permite a abstração por parte do programador sobre como os dados são enviados e recebidos.

Segue um exemplo de mensagem enviada pela simulação do mundo para o agente.

```
{
  "head_angle": 0.0,
  "objects_list": [
    {
      "kind": "ball",
      "position": [
        239.9031999919439,
        -0.6876952725551101
      ]
    }
  ]
}
```

Neste exemplo pode-se perceber as informações enviadas pelo ambiente ao agente. O primeiro atributo, *head_angle*, indica o ângulo da cabeça do robô em relação ao corpo. O próximo atributo, *objects_list*, é um *array* contendo os objetos presentes no mundo.

No exemplo, é passado somente um objeto, o que não significa que há somente um objeto no ambiente. Para que a comunicação com um agente seja feita é pré-requisito que um robô esteja presente no ambiente que, somando com a bola, totaliza dois objetos. O robô com o qual o agente está se comunicando não é adicionado na lista de objetos e, por esse motivo, apenas uma bola é listada no exemplo.

Além disso, mais objetos poderiam estar presentes no ambiente mas fora do campo de visão do robô e a simulação do ambiente verifica isso antes de montar a lista de objetos a serem enviados para o agente.

Cada objeto é representado somente por um atributo *kind*, que neste caso é uma bola, e um atributo *position* que é um *array* de duas posições representando as coordenadas r e θ.

O interessante dessa abordagem é que ela pode ser aplicada ao robô real fornecendo um formato padrão para a comunicação entre os sensores e a mente do robô. A parte da visão no robô real pode informar os dados sobre os objetos percebidos no ambiente em formato *JSON* possibilitando a portabilidade do código desenvolvido para testar a estratégia no simulador diretamente para o robô real. Como o formato de dados é o mesmo, não existe necessidade de adaptar o código para receber os dados vindos dos sensores.

4.5 *Controller*

O módulo *Controller* possui uma única classe de mesmo nome.

A classe *Controller* guarda em seus atributos uma referência do modelo de mundo, a

visão e os comunicadores relativos aos robôs presentes no mundo. Além disso é no controlador que está a definição de como o sistema reage a comandos do usuário, por isso ele guarda a posição do cursor, o objeto que está sendo movido com o mouse, caso haja um, e a posição relativa entre a posição do clique e do centro do objeto a ser movido.

Em seus métodos estão definidas as ações a serem executadas em resposta às interações do usuário e o controle da simulação e da comunicação entre ambiente e agentes. O método *updateSimulation* ouve eventos do *pygame*, faz a comunicação e atualiza a tela através do método *update* da *view*.

Para fazer a comunicação o controlador faz uma iteração por todos os comunicadores criados até o momento e, caso alguma mensagem tenha sido recebida a partir de um agente, ele repassa o comando ao robô que aquele agente está tentando controlar. Após atualizar as posições dos objetos do mundo de acordo com os comandos recebidos dos agentes, ele envia ao agente uma mensagem informando sobre o ambiente percebido.

4.6 Sumário do Capítulo

O simulador funciona com duas partes que se comunicam via rede através de objetos *JSON*. Uma das partes, é a simulação do ambiente onde o usuário irá criar o cenário no qual deseja testar o comportamento da sua estratégia. O funcionamento desta primeira parte foi descrito até aqui, passemos agora ao funcionamento da parte que serve como camada de abstração para o usuário.

5 MODELAGEM: SIMULAÇÃO DO ROBÔ

A simulação do ambiente só passa a ser útil quando um agente conecte-se a ela. Enquanto nenhum agente está conectado é como se nenhum robô presente no ambiente tivesse uma mente.

A simulação do robô é útil para o agente porque é ela quem recebe os dados percebidos no ambiente e interpreta criando uma abstração útil para o programador da inteligência artificial.

A simulação do robô ainda pode ser acoplada no funcionamento do robô real, funcionando como uma camada de abstração para o programador. Se o sistema de visão identificar objetos no ambiente e informar à inteligência artificial sobre esses objetos através de objetos no formato *JSON*, o código desenvolvido para o agente irá funcionar sem necessidade de adaptações.

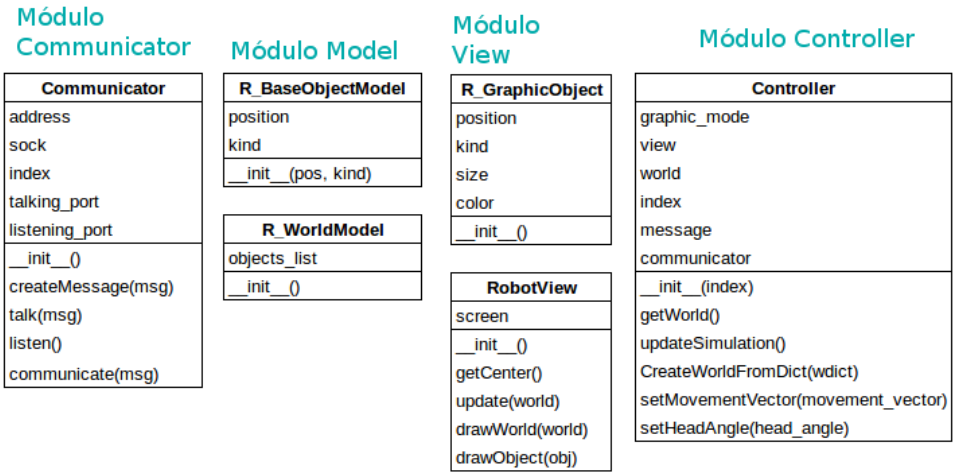

Figura 5.1: Módulos de *Robot*

O usuário ainda pode visualizar a perspectiva do robô sendo controlado pelo agente conectado ao sistema através de uma *view*. A parte que simula o robô, portanto, possui os mesmos módulos que a simulação do mundo.

5.1 *Model* e *View*

As classes pertencentes aos módulos *Model, View, Communicator* e *Controller* têm funcionamento muito similar às que pertencem aos módulos de mesmo nome da simulação do

ambiente. É possível ver na Figura 5.1 que as classes referentes ao modelo dos objetos e do mundo são mais simples pois o robô tem menos informações sobre o ambiente do que a simulação do próprio ambiente.

A *view* também é uma versão simplificada da *view* do mundo porque a visão da perspectiva do robô trata menos informações sobre os objetos do que a visão do mundo. A incerteza de um objeto, por exemplo, é desconhecida pelo robô porque ele recebe apenas uma posição. Essa posição é gerada dentro do raio de incerteza e não necessariamente condiz com a posição real do objeto.

5.2 *Communicator*

O comunicador do robô recebe e envia dados ao mundo. Em uma situação real o comunicador pode ler dados dos sensores e escrever dados para os atuadores. O robô real precisa ter sua programação de mais baixo nível adaptada para receber comandos no formato *JSON* mas, uma vez que essa parte seja capaz de interpretar esses comandos, a programação da inteligência artificial do robô pode receber e enviar dados de uma maneira muito mais simples.

O usuário do simulador não escreve diretamente os objetos *JSON* mas pode ter controle sobre as mensagens recebidas e enviadas pelo comunicador através da classe *controller*, que será descrita em seguida.

A forma com a qual são decididos o endereço e as portas de rede para comunicação de dados seguem o mesmo padrão descrito na Seção 4.3.

A seguir, um exemplo de mensagem enviada pela parte que simula a percepção do robô ao ambiente através de comandos dados pelo programador do agente.

```
{
   "movement_vector": [
      5,
      -0.9758561608856633,
      -0.9758561608856633
   ],
   "index": 0,
   "head_angle": -0.9758561608856633
}
```

Na mensagem estão presentes um *movement_vector* que indica o movimento a ser feito pelo robô no ambiente através de um vetor de três coordenadas do tipo (r, θ, ϕ) onde r e θ são componentes polares indicando um movimento e ϕ é um ângulo de rotação para que o robô gire em seu próprio eixo. Seguindo a representação apresentada anteriormente, ângulos à esquerda são representados por valores entre 0 e π e valores à direita entre 0 e $-\pi$.

O valor de *index* informa ao ambiente qual robô o agente quer controlar e *head_angle* informa a rotação que deve ser feita na cabeça do robô. O ângulo da cabeça é informado como um valor absoluto, ou seja, o simulador gera o incremento necessário para que aquele valor seja alcançado. Já o ângulo representado por ϕ indica um incremento na rotação do corpo do robô.

Isso acontece pois, para o robô, ele está sempre no ângulo zero já que ele é a origem do seu sistema de coordenadas. Após rotacionar seu corpo em um certo ângulo essa nova posição é considerada sua posição zero. Já o ângulo da cabeça é representado em relação ao corpo do robô e por isso pode ser representado em ângulos absolutos.

5.3 *Controller*

O controlador da simulação do robô é, a exemplo dos outros módulos, uma versão mais simples de seu módulo correspondente em *world*. Na classe *controller* o atributo *graphic_mode* indica se a *view* será utilizada e, caso seja, uma tela irá exibir a perspectiva do robô. Os atributos *view*, *world* e *communicator* foram explicados anteriormente e o atributo *index* indica a qual robô presente no ambiente esta simulação está conectada.

Quando o usuário programa sua estratégia em *python* ele pode usar a classe *controller* como *API*. Por isso o controlador oferece o atributo *message* que permite que o usuário leia e escreva diretamente o objeto *JSON* a ser enviado. Além disso ele pode acessar a lista de objetos presentes no ambiente através do método *getWorld*. A instância de *world* é recriada cada vez que o método *updateSimulation* é executado. Este método utiliza o método *createWorldFromDict* para recriar o mundo a partir da mensagem recebida do ambiente.

Os métodos *setMovementVector* e *setHeadAngle* permitem ao usuário setar valores para o vetor de movimento e para o ângulo da cabeça do robô sem alterar diretamente a mensagem a ser enviada ao ambiente.

5.4 O paradoxo de Teseu

Para otimizar o processamento do sistema poderíamos apenas remover objetos da lista de objetos conhecidos pelo robô quando eles saíssem do campo de visão do robô e adicionar objetos à lista quando eles entrassem no campo de visão do robô, ao invés de esvaziar completamente a lista e criá-la novamente. O paradoxo de Teseu ajuda a justificar a motivação que serve como embasamento para a abordagem de recriar a lista.

A lenda grega conta o seguinte:

> O navio com que Teseu e os jovens de Atenas retornaram (de Creta) tinha trinta remos, e foi preservado pelos atenienses até o tempo de Demétrio de Falero, porque eles removiam as partes velhas que apodreciam e colocavam partes novas, de forma que o navio se tornou motivo de discussão entre os filósofos a respeito do conjunto de caracteres próprios e exclusivos com os quais se podem diferenciar objetos inanimados uns dos outros (REA, 1995).

A pergunta levantada pelos filósofos era: se o navio tem suas partes modificadas periodicamente ao ponto que toda a madeira nele tenha sido substituída, até que ponto podemos dizer que esse ainda é o mesmo navio?

Quando o robô recebe os dados enviados pelo ambiente não podemos ter certeza que um objeto presente no campo de visão é o mesmo que o robô percebeu no laço anterior, mesmo se estiver na mesma posição. Se o robô percebe, por exemplo, um outro robô em uma posição A e no próximo laço ele percebe dois robôs, um em uma posição B e outro em um posição C, não podemos indicar com certeza se o robô se moveu de A para B ou de A para C. Os dois novos robôs talvez sejam, inclusive, diferentes daquele na posição A enquanto o que estava na posição A saiu do campo de visão.

Alguma demora no processamento do robô real pode fazer com que um laço leve mais tempo do que o esperado e as posições dos objetos podem se alterar drasticamente entre um laço e outro. Por todos esses motivos foi decidido que a abordagem mais segura seria recriar toda a lista de objetos.

5.5 Sumário do Capítulo

O sistema oferece uma simulação do robô que serve como abstração para o programador e oferece uma *API* para acessar as informações fornecidas pelo ambiente. A simulação do robô pode ter uma *view* com a perspectiva do mundo percebido pelo robô e possui funcionamento similar à simulação do ambiente.

6 CONCLUSÃO

No ano de 2015 a equipe *Taura Bots*, representante da UFSM, participará da *RoboCup* e para desenvolver a estratégia de comportamento do seu robô foi criado um simulador computacional.

O problema abordado foi se seria possível criar um simulador computacional aplicado a futebol de robôs que criasse um alto nível de abstração sem abrir mão da compatibilidade com o hardware. De certa maneira, a programação do robô real precisaria de várias camadas de abstração para que a programação de alto nível fosse realmente transparente, uma vez que alterações precisam ser feitas no código do robô para que a abordagem usada durante este trabalho funcione. Apesar disso, a única adaptação necessária é na forma com a qual é feita a comunicação de dados do robô real.

Considerando o sistema desenvolvido durante a execução deste trabalho, os objetivos alcançados foram: criar um ambiente de simulação de forma que o desenvolvimento de uma estratégia de comportamento não tenha como requisito a disponibilidade do robô real; e criar um nível de abstração conveniente para o desenvolvimento de estratégias de comportamento escondendo do programador a forma com a qual se dá o funcionamento de algumas partes do sistema do robô. O objetivo de criar um ambiente de simulação de maneira tal que o código desenvolvido no simulador possa ser usado diretamente no robô sem a necessidade de grandes alterações fica dependente de testes uma vez que uma estratégia de comportamento não foi totalmente desenvolvida, dado o nível que a Taura Bots ainda é iniciante na competição.

Apesar disso, o conceito por trás do simulador se mostrou extremamente eficaz e tem futuro promissor levando em consideração que a equipe da UFSM tem como objetivo participar da *RoboCup* nos próximos anos utilizando, inclusive, robôs maiores. Com o aperfeiçoamento do sistema e um prazo maior para o desenvolvimento dos robôs, uma estratégia de comportamento pode ser desenvolvida inteiramente através do sistema de simulação e o sistema do robô desenvolvido pode ser adaptado desde estágios iniciais para receber os códigos desenvolvidos no ambiente de testes.

O código do sistema desenvolvido durante a execução deste trabalho encontra-se em anexo para fins de consultas do leitor, mesmo que o sistema desenvolvido não tenha servido como fim para a pesquisa sobre a abordagem apresentada neste trabalho, mas sim como meio para o teste da mesma. Vários simuladores diferentes podem ser desenvolvidos usando diversas

linguagens de programação, mas o ponto central da abordagem apresentada neste trabalho é a utilização do formato JSON para comunicação de dados criando uma abstração de tal forma que o desenvolvimento de estratégias no simulador seja transparente para o desenvolvedor.

REFERÊNCIAS

BANKS, J.; CARSON, J.; NELSON, B. **DM Nicol, Discrete-Event System Simulation**. [S.l.]: Prentice hall Englewood Cliffs, NJ, USA, 2000.

CAMPBELL, M.; HOANE, A. J.; HSU, F.-h. Deep blue. **Artificial intelligence**, [S.l.], v.134, n.1, p.57–83, 2002.

CROCKFORD, D. The application/json media type for JavaScript Object Notation (JSON), 2006. **URL http://tools. ietf. org/html/rfc4627**, [S.l.], 2013.

ITSUKI, N. Soccer server: a simulator for robocup. In: JSAI AI-SYMPOSIUM 95: SPECIAL SESSION ON ROBOCUP. **Anais...** [S.l.: s.n.], 1995.

KITANO, H. et al. RoboCup: a challenge problem for ai and robotics. **RoboCup-97: Robot Soccer World Cup I**, [S.l.], v.18, n.1, p.1–19, 1998.

KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: INTELLIGENT ROBOTS AND SYSTEMS, 2004.(IROS 2004). PROCEEDINGS. 2004 IEEE/RSJ INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2004. v.3, p.2149–2154.

KOENIG, N.; HOWARD, A. Gazebo-3D multiple robot simulator with dynamics (2003). **URL: http://gazebosim. org**, [S.l.], v.3, 2013.

PFEIFER, R.; SCHEIER, C.; ILLUSTRATOR-FOLLATH, I. **Understanding intelligence**. [S.l.]: MIT press, 2001.

REA, M. C. The problem of material constitution. **The Philosophical Review**, [S.l.], p.525–552, 1995.

RUSSELL, S.; NORVIG, P.; INTELLIGENCE, A. A modern approach. **Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs**, [S.l.], v.25, 1995.

SANNER, M. F. et al. Python: a programming language for software integration and development. **J Mol Graph Model**, [S.l.], v.17, n.1, p.57–61, 1999.

SHINNERS, P. **Pygame**. 2011.

ANEXOS

ANEXO A – Código fonte do simulador

O código fonte a seguir apresentado é disponibilizado apenas para a análise por parte do leitor, portanto somente módulos e partes citadas no texto estão anexadas, não incluindo arquivos de configuração do sistema e alguns trechos de código. Por esse motivo, o código a seguir não condiz com uma versão funcional do sistema.

```
##### libs/Vector2.py #####
class Vector2(object):
   def __init__(self, x=None, y=None, phi=None, r=None, a=None,
      rectCoords=None, polarCoords=None, vector=None):

      self.__dict__['x'] = 0
      self.__dict__['y'] = 0
      self.__dict__['r'] = 0
      self.__dict__['a'] = 0
      self.__dict__['phi'] = 0
      if not x==None and not y==None:
         self.__dict__['x'] = x
         self.__dict__['y'] = y
         self.__dict__['r'] = sqrt(self.x**2 + self.y**2)
         self.__dict__['a'] = atan2(self.y, self.x)
      elif not r==None and not a==None:
         self.__dict__['r'] = r
         self.__dict__['a'] = a
         self.__dict__['x'] = self.r * cos(self.a)
         self.__dict__['y'] = self.r * sin(self.a)
      elif rectCoords:
         self.__dict__['x'] = rectCoords[0]
         self.__dict__['y'] = rectCoords[1]
         self.__dict__['r'] = sqrt(self.x**2 + self.y**2)
         self.__dict__['a'] = atan2(self.y, self.x)
      elif polarCoords:
         self.__dict__['r'] = polarCoords[0]
         self.__dict__['a'] = polarCoords[1]
         self.__dict__['x'] = self.r * cos(self.a)
         self.__dict__['y'] = self.r * sin(self.a)
      elif vector:
         self.__dict__['x'] = vector.x
         self.__dict__['y'] = vector.y
         self.__dict__['r'] = vector.r
         self.__dict__['a'] = vector.a
         self.__dict__['phi'] = vector.phi
      if phi:
         self.__dict__['phi'] = phi

   def recalc(self, fromRect=True):
      if fromRect:
```

```
            self.__dict__['r'] = sqrt(self.x**2 + self.y**2)
            self.__dict__['a'] = atan2(self.y, self.x)
        else:
            self.__dict__['x'] = self.r * cos(self.a)
            self.__dict__['y'] = self.r * sin(self.a)
        return self

    def setCoords(self, coords, isRect=True):
        if isRect:
            self.x = coords[0]
            self.y = coords[1]
            self.recalc()
        else:
            self.r = coords[0]
            self.a = coords[1]
            self.recalc(False)
        return self

    def setX(self, x):
        self.x = x
        self.recalc()
        return self

    def setY(self, y):
        self.y = y
        self.recalc()
        return self

    def setR(self, r):
        self.r = r
        self.recalc()
        return self

    def setA(self, a):
        self.a = a
        self.recalc()
        return self

    def getCoords(self, isRect=True, getPhi=False):
        if isRect:
            if getPhi:
                return (self.x, self.y, self.phi)
            return (self.x, self.y)
        else:
            if getPhi:
                return (self.r, self.a, self.phi)
            return (self.r, self.a)

    def zero(self):
        self.__dict__['x'] = 0
        self.__dict__['y'] = 0
```

```
        self.__dict__['r'] = 0
        self.__dict__['a'] = 0
        self.__dict__['phi'] = 0

    def __setattr__(self, name, value):
        if name == 'x' or name == 'y':
            self.__dict__[name] = value
            self.recalc()
        elif name == 'r' or name == 'a':
            self.__dict__[name] = value
            self.recalc(False)
        else:
            self.__dict__[name] = value

    def __iadd__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector2(x,y)

    def __isub__(self, other):
        x = self.x - other.x
        y = self.y - other.y
        return Vector2(x,y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector2(x,y)

    def __sub__(self, other):
        x = self.x - other.x
        y = self.y - other.y
        return Vector2(x,y)

    def __mul__(self, other):
        r = self.__dict__['r'] = self.r * other
        a = self.a
        return Vector2(r=r, a=a)

    def __len__(self):
        return int(self.r)

    def __repr__(self):
        return "r({0}, {1}) p({2}, {3})".format(str(self.x),
            str(self.y), str(self.r), str(self.a))

    def normalize(self):
        self.__dict__['x'] = self.x / self.r
        self.__dict__['y'] = self.y / self.r
        self.recalc()
```

```
##### World/Model.py #####
class W_BaseObjectModel(object):
   def __init__(self, pos = Vector2(0, 0), kind = UNKNOWN):
      self.position = pos
      self.kind = kind
      self.uncertainty = 30

   def __repr__(self):
      return ("{1}:{{'position': ({0}), 'uncertainty':
         {2}}}".format(self.position, self.kind, self.uncertainty))

class W_RobotModel(W_BaseObjectModel):
   def __init__(self, pos, kind, world):
      super().__init__(pos, kind)
      self.head_angle = 0.0
      self.body_angle = 0.0
      self.head_angle_step = pi/18
      self.body_angle_step = pi/18
      self.head_angle_limit = pi/2
      self.index = 0
      self.movement_vector = Vector2(0.0, 0.0, 0.0)
      self.vision_width = pi/2
      self.world = world
      self.kicking = None

   def setMovementVector(self, movement_vector):
      self.movement_vector = movement_vector
      self.move()

   def setHeadAngle(self, angle):
      # limita o angulo para que seja um angulo valido
      angle = limitAngle(angle, self.head_angle_limit)
      # calcula o incremento necessario para chegar ao angulo
         desejado
      increment = angle - self.head_angle
      # limita o incremento para que seja no maximo do tamanho do
         passo
      increment = limitAngle(increment, self.head_angle_step)

      self.head_angle += increment

   def move(self):
      phi = self.movement_vector.phi

      self.body_angle += limitAngle(phi, self.body_angle_step)

      new_position = self.position + self.movement_vector

      if not self.world.checkCollision(new_position, ROBOT_RADIUS,
         self):
         self.position = new_position
```

```
    def setKick(self, vector):
        self.kicking = vector

class W_WorldModel:
    def __init__(self):
        self.objects_list = []
        self.robots_list = []

    def getRobots(self):
        robots = []
        for obj in self.objects_list:
            if obj.kind == ROBOT:
                robots.append(obj)
        return robots

    def checkCollision(self, pos, r=0, check_obj=None):
        for obj in self.objects_list:
            if obj == check_obj:
                pass
            else:
                dist = (pos - obj.position).r
                radius = ROBOT_RADIUS if obj.kind == ROBOT else
                    OBJ_RADIUS
                if dist < radius + r:
                    return obj
        return None

    def getEgocentric(self, robot):
        obj_list = self.objects_list
        final_list = []
        for obj in obj_list:
            # checando objeto
            if obj != robot:
                # nao eh meu robo
                # gera ruido de maneira polar
                unc_pos = generateUncertainty(obj.uncertainty)

                # soma a posicao do objeto ao ruido
                obj_pos = obj.position + unc_pos

                # passa a posicao com o ruido para um sistema onde o
                    robo eh a origem
                # essa eh a posicao relativa
                rel_pos = obj_pos - robot.position

                # soma o angulo da cabeca + angulo do corpo
                robot_angle = robot.head_angle + robot.body_angle
                angle = rel_pos.a + robot_angle

                # inverte o angulo porque com o Y aumentando pra baixo o
```

```
                angulo aumenta em sentido anti-horario
            angle = angle * -1

            # normaliza o angulo
            # 0 ~ 2PI => PI ~ -PI
            angle = normalizeAngle(angle)

            limit_left = -robot.vision_width/2
            limit_right = robot.vision_width/2

            if angle > limit_left and angle < limit_right:
                rel_pos.a = angle
                new_obj = copy.deepcopy(obj)
                new_obj.position = rel_pos

                final_list.append(new_obj)

        return final_list

##### World/View.py #####

class W_GraphicObject:
    def __init__(self, obj):
        self.position = obj.position.getCoords()
        self.uncertainty = obj.uncertainty
        self.kind = obj.kind

        self.size = 5
        self.color = Color('white')
        self.head_angle = 0.0
        self.body_angle = 0.0
        self.movement_vector = (0, 0)

        if obj.kind == UNKNOWN:
            self.size = 10
            self.color = Color('#000000')
        elif obj.kind == BALL:
            self.size = 10
            self.color = Color('#FFFFFF')
        elif obj.kind == POLE:
            self.size = 10
            self.color = Color('#EE9900')
        elif obj.kind == ROBOT:
            self.size = 20
            self.color = Color('#0099EE')
            self.head_abs_angle = obj.head_angle + obj.body_angle
            self.body_angle = obj.body_angle
            self.movement_vector = obj.movement_vector
            self.vision_width = obj.vision_width

class WorldView:
```

```
def __init__(self):
    self.screen = {
        'width': WIDTH,
        'height': HEIGHT,
        'surface': None,
        'background': Color(0, 122, 0)
    }
    self.help = True
    pygame.display.set_caption('Robocup Simulator - World')
    self.screen['surface'] = pygame.display.set_mode(
        (self.screen['width'], self.screen['height']), 0, 32)
    self.mouse_position = (0, 0)

def setMousePosition(self, pos):
    self.mouse_position = pos

def update(self, world):
    self.drawField()
    self.drawWorld(world)
    if self.help:
        self.drawHelp()
    pygame.display.flip()

def blitSurface(self, surface, position):
    self.screen['surface'].blit(surface, position)

def drawHelp(self):
    font = pygame.font.SysFont("monospace", 15)
    help_string = [
        "r - insert robot",
        "b - insert ball",
        "p - insert pole",
        "o - insert unknown object",
        "h - show/hide this help message",
        "q - quit",
        "left/right - rotates the robot's body",
        "a/d - rotates the robot's head",
        "scroll - increase/decrease object's uncertainty"]

    for i, string in enumerate(help_string):
        surf = font.render(string, True, (0,0,0))
        self.blitSurface(surf, (30, 15 + i*15))

    font = pygame.font.SysFont("monospace", 10)

    h = self.screen['height']
    for i in range(0, h, 25):
        surf = font.render(str(i), True, (0,0,0))
        self.blitSurface(surf, (0, i - 10))

    for i in range(0, self.screen['width'], 25):
```

```
            surf = font.render(str(i), True, (0,0,0))
            self.blitSurface(surf, (i, h - 10))

    def drawField(self):
        self.screen['surface'].fill(self.screen['background'])

        h_poly_pts = [
            (0, self.mouse_position[1] + 2),
            (0, self.mouse_position[1] - 2),
            (5, self.mouse_position[1])
        ]
        draw.polygon(self.screen['surface'], (0,0,0), h_poly_pts)

        w_poly_pts = [
            (self.mouse_position[0] + 2, self.screen['height']),
            (self.mouse_position[0] - 2, self.screen['height']),
            (self.mouse_position[0], self.screen['height'] - 5)
        ]
        draw.polygon(self.screen['surface'], (0,0,0), w_poly_pts)

    def drawWorld(self, world):
        for obj in world.objects_list:
            self.drawObject(W_GraphicObject(obj))

    def drawObject(self, obj):
        obj_area = obj.size*2
        unc_area = (obj.uncertainty * 2) + obj_area

        unc_surface = pygame.Surface((unc_area, unc_area))
        unc_surface.fill(self.screen['background'])
        unc_surface.set_colorkey(self.screen['background'])
        unc_surface.set_alpha(122)
        unc_surface_center = (int(unc_area/2), int(unc_area/2))
        pygame.draw.circle(unc_surface, obj.color, unc_surface_center,
            int(unc_area/2))

        obj_surface = pygame.Surface((obj_area, obj_area))
        obj_surface.fill(self.screen['background'])
        obj_surface.set_colorkey(self.screen['background'])
        obj_surface_center = (obj.size, obj.size)
        rect = pygame.draw.circle(obj_surface, obj.color,
            obj_surface_center, obj.size)


        if obj.kind == ROBOT:
            # desenhar orientacao da cabeca
            end_pos = toRectangular( (obj.size, - obj.head_abs_angle) )
            end_pos = (
                obj_surface_center[0] + end_pos[0],
                obj_surface_center[1] + end_pos[1]
            )
```

```
pygame.draw.line(obj_surface, (255,0,0),
   obj_surface_center, end_pos, 3)
# desenhar parte preta que representa as costas do robo
pygame.draw.arc(obj_surface, (0,0,0), rect, obj.body_angle
   + pi/2, obj.body_angle + 3*pi/2, obj.size)

# desenha o vetor de movimento
movement_size = 25


obj.movement_vector = obj.movement_vector.getCoords()


movement_point =(
   obj_surface_center[0] + obj.movement_vector[0] / 1 *
      movement_size,
   obj_surface_center[1] + obj.movement_vector[1] / 1 *
      movement_size
)
pygame.draw.line(obj_surface, (255,0,255),
   obj_surface_center, movement_point, 3)

# campo de visao
vision_width = obj.vision_width
vision_range = 100

vision_surface = pygame.Surface((vision_range * 2,
   vision_range * 2))
vision_surface.fill(self.screen['background'])
vision_surface.set_colorkey(self.screen['background'])
vision_surface.set_alpha(200)
vision_surface_center = (vision_range, vision_range)

angle_1 = - obj.head_abs_angle + vision_width/2
angle_2 = - obj.head_abs_angle - vision_width/2

point_1 = (vision_range, angle_1)
point_2 = (vision_range, angle_2)

point_1 = toRectangular(point_1)
point_2 = toRectangular(point_2)

point_1 = (point_1[0] + vision_surface_center[0],
   point_1[1] + vision_surface_center[1])
point_2 = (point_2[0] + vision_surface_center[0],
   point_2[1] + vision_surface_center[1])

pygame.draw.line(vision_surface, (120,120,120),
   vision_surface_center, point_1, 5)
pygame.draw.line(vision_surface, (120,120,120),
   vision_surface_center, point_2, 5)
```

```
            position = (
               obj.position[0] - vision_range,
               obj.position[1] - vision_range
            )
            self.blitSurface(vision_surface, position)

         position = (
            obj.position[0]-unc_area/2,
            obj.position[1]-unc_area/2
         )
         self.blitSurface(unc_surface, position)

         position = (
            obj.position[0]-obj.size,
            obj.position[1]-obj.size
         )
         self.blitSurface(obj_surface, position)

##### World/Communicator.py #####

class Communicator:
   def __init__(self, index, world):
      self.address = LOCALHOST
      self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
      self.index = index
      self.listening_port = INITIAL_PORT + (index * 2)
      self.talking_port = self.listening_port + 1
      self.sock.bind((self.address, self.listening_port))

      self.world = world

   def isThisObjMyRobot(self, obj):
      if obj.kind != ROBOT:
         return False
      elif obj.index == self.index:
         return True
      else:
         return False

   def createMessage(self):
      world = {
         'objects_list': []
      }
      my_robot = self.world.getRobots()[self.index]
      obj_list = self.world.getEgocentric(my_robot)

      for obj in obj_list:
         obj_dict = {
            'position': obj.position.getCoords(False),
            'kind': obj.kind
```

```
            }
            world['objects_list'].append(obj_dict)

        world['head_angle'] = my_robot.head_angle
        # print("mundo enviando:", world)
        return json.dumps(world)


    def talk(self):
        message = self.createMessage()
        print("World sending:", message)
        self.sock.sendto(bytes(message, 'UTF-8'), (self.address,
            self.talking_port))

    def listen(self):
        # print('Server listening...')
        try:
            self.sock.settimeout(1/100)
            data, addr = self.sock.recvfrom(1024)
            message = data.decode('UTF-8')
            # print('Received message: ', message)
            return json.loads(message)
        except socket.error:
            # print(socket.error.errno)
            pass
        return False

##### World/Controller #####

class Controller:
    def __init__(self):
        self.world = W_WorldModel()
        self.view = WorldView()
        self.communicators = []

        self.mouse_position = Vector2(0, 0)
        self.moving_obj = None
        self.rel_click_pos = Vector2(0, 0)

    def removeObject(self):
        clicked_obj = self.world.checkCollision(self.mouse_position)
        if clicked_obj in self.world.objects_list:
            index = self.world.objects_list.index(clicked_obj)
            del self.world.objects_list[index]

    def setMovingObject(self):
        clicked_obj = self.world.checkCollision(self.mouse_position)
        if clicked_obj:
            self.moving_obj = clicked_obj
            obj_pos = clicked_obj.position
            click_pos = self.mouse_position
```

```
            self.rel_click_pos = click_pos - obj_pos
        else:
            self.moving_obj = None


    def updateSimulation(self):
        for event in pygame.event.get():
            if event.type == QUIT:
                return False
            else:
                if not self.handleEvent(event):
                    return False

        self.communicate()

        self.view.update(self.world)
        return True

    def communicate(self):
        for communicator in self.communicators:
            message = communicator.listen()
            if message:
                i = message['index']
                mv_tuple = (message['movement_vector'][0],
                    message['movement_vector'][1])
                mv_phi = message['movement_vector'][2]

                mv_vector = Vector2(polarCoords=mv_tuple)

                mv_vector.phi = mv_phi
                hd_angle = message['head_angle']

                robot = self.world.robots_list[i]
                # corrigir o angulo do movimento de acordo com o angulo
                    do corpo
                body_angle = self.world.robots_list[i].body_angle
                mv_vector.a += body_angle
                # inverte o Y
                mv_vector.a = mv_vector.a * -1

                robot.setMovementVector(mv_vector)
                robot.setHeadAngle(hd_angle)
                if "kick" in message:
                    robot.setKick(message['kick'])
                else:
                    robot.setKick(None)

                communicator.talk()

    def checkMovingObject(self):
        if self.moving_obj:
```

```
            button_pressed,_,_ = pygame.mouse.get_pressed()
            if button_pressed:
                new_position = self.mouse_position - self.rel_click_pos
                radius = OBJ_RADIUS
                if self.moving_obj.kind == ROBOT:
                    radius = ROBOT_RADIUS
                if not self.world.checkCollision(new_position, radius,
                    self.moving_obj):
                    self.moving_obj.position = new_position
            else:
                self.moving_obj = None


    def handleEvent(self, event):
        if debug: print("event.type:", event)

        if event.type == MOUSEMOTION:
            self.mouse_position =
                Vector2(rectCoords=pygame.mouse.get_pos())
            self.view.setMousePosition(self.mouse_position.getCoords())
            self.checkMovingObject()
        elif event.type == MOUSEBUTTONDOWN:
            if event.button == 1: self.setMovingObject() # left click
            elif event.button == 3: self.removeObject() # right click
            elif event.button == 5: # scroll UP
                obj = self.getClosestObject()
                obj.uncertainty += 1
            elif event.button == 4: # scroll DOWN
                obj = self.getClosestObject()
                if obj.uncertainty -1 >= 0:
                    obj.uncertainty -= 1
        elif event.type == KEYDOWN:
            # create object
            if event.key in objectKeys.keys():
                self.createObject(event.key)
            # control closest robot
            elif event.key in controlKeys:
                self.controlRobot(self.getClosestRobot(), event.key)
            # show / hide help
            elif event.key == K_h:
                self.view.help = not self.view.help
            # quit
            elif event.key == K_q:
                return False
        elif event.type==VIDEORESIZE:
            screen=pygame.display.set_mode(event.dict['size'],
                pygame.RESIZABLE, 32)
        return True


    def createObject(self, key):
```

```
        new_object = None
        mouse_vector = self.mouse_position
        if key == K_r:
            if not self.world.checkCollision(mouse_vector,
                ROBOT_RADIUS):
                new_object = W_RobotModel(mouse_vector, ROBOT,
                    self.world)
                robot_index = len(self.world.getRobots())
                new_object.index = robot_index
                self.communicators.append(Communicator(robot_index,
                    self.world))
                self.world.robots_list.append(new_object)
        elif not self.world.checkCollision(mouse_vector, OBJ_RADIUS):
            new_object = W_BaseObjectModel(mouse_vector,
                objectKeys[key])
        if new_object:
            self.world.objects_list.append(new_object)

    def getClosestObject(self):
        lowest_distance = 999999999.9
        closest_object = None
        for obj in self.world.objects_list:
            distance = (obj.position - self.mouse_position).r
            if distance < lowest_distance:
                closest_object = obj
                lowest_distance = distance
        return closest_object

    def getClosestRobot(self):
        lowest_distance = 999999999.9
        closest_robot = None
        for obj in self.world.objects_list:
            if obj.kind == ROBOT:
                distance = (obj.position - self.mouse_position).r
                if distance < lowest_distance:
                    closest_robot = obj
                    lowest_distance = distance
        return closest_robot

    def controlRobot(self, robot, key):
        if robot:
            if key == K_RIGHT:
                robot.body_angle -= robot.body_angle_step
                # print(degrees(robot.body_angle))
                print("body:", degrees(robot.body_angle), "head:",
                    degrees(robot.head_angle))
            elif key == K_LEFT:
                robot.body_angle += robot.body_angle_step
                # print(degrees(robot.body_angle))
                print("body:", degrees(robot.body_angle), "head:",
                    degrees(robot.head_angle))
```

```
            elif key == K_UP:
               pass
            elif key == K_DOWN:
               pass
            elif key == K_d:
               if (robot.head_angle - robot.head_angle_step) >= -(pi/2):
                  robot.head_angle -= robot.head_angle_step
            elif key == K_a:
               if (robot.head_angle + robot.head_angle_step) <= (pi/2):
                  robot.head_angle += robot.head_angle_step

            if debug: print("body:", robot.body_angle, "head:",
               robot.head_angle)

##### Robot/Model.py #####

class R_BaseObjectModel(object):
   def __init__(self, pos, kind = UNKNOWN):
      self.position = pos
      self.kind = kind

   def getJSON(self):
      return json.dumps(self, default=lambda o: o.__dict__)

class R_WorldModel:
   def __init__(self):
      self.objects_list = []

   def getJSON(self):
      return json.dumps(self, default=lambda o: o.__dict__)#,
         sort_keys=True, indent=4)

   def getDict(self):
      return json.loads(self.toJSON())

##### Robot/View.py #####
class R_GraphicObject:
   def __init__(self, obj):
      self.position = obj.position
      self.kind = obj.kind
      if obj.kind == UNKNOWN:
         self.size = 10
         self.color = Color('#000000')
      elif obj.kind == BALL:
         self.size = 10
         self.color = Color('#FFFFFF')
      elif obj.kind == POLE:
         self.size = 10
         self.color = Color('#EE9900')
      elif obj.kind == ROBOT:
         self.size = 20
```

```
            self.color = Color('#0099EE')

class RobotView:
    def __init__(self):
        self.screen = {
            'width': 500,
            'height': 500,
            'surface': None,
            'center': (int(500/2), int(500/2))
        }
        pygame.init()
        self.clock = pygame.time.Clock()
        pygame.display.set_caption('Robocup Simulator - Robot')
        self.screen['surface'] = pygame.display.set_mode(
            (self.screen['height'], self.screen['width']), 0, 32)

    def getCenter(self):
        return self.screen['center']

    def update(self, world):
        self.drawField()
        self.drawWorld(world)

        pygame.display.flip()
        self.clock.tick(60)

    def drawField(self):
        self.screen['surface'].fill((0,0,122))

    def drawWorld(self, world):
        # desenha uma circunferencia representando o proprio robo
        pygame.draw.circle(self.screen['surface'], Color('#0099EE'),
            (self.getCenter()[0], self.screen['height']), 20)
        for obj in world.objects_list:
            self.drawObject(R_GraphicObject(obj))

    def drawObject(self, obj):
        # gira 90 graus para ficar com o 0 para frente
        # -90 graus pq com o y crescendo pra baixo o angulo aumenta
            anti-horario
        angle = obj.position[1]
        angle *= -1
        angle -= pi/2
        # converte para retangulares para exibir na tela
        pos = toRectangular((obj.position[0], angle))
        # translada da origem para o centro da tela
        pos = (self.getCenter()[0] + int(pos[0]),
            self.screen['height'] + int(pos[1]))

        pygame.draw.circle(self.screen['surface'], obj.color, pos,
            obj.size)
```

```
##### Robot/Conmmunicator.py #####
lass Communicator:

   def __init__(self, index = 0):
      self.address = LOCALHOST
      self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
      self.index = index
      self.talking_port = INITIAL_PORT + (index * 2)
      self.listening_port = self.talking_port + 1

      self.sock.bind((self.address, self.listening_port))

   def createMessage(self, msg):
      message = {
         'index': self.index,
         'movement_vector': msg['movement_vector'].getCoords(False,
            True),
         'head_angle': msg['head_angle']
      }
      return json.dumps(message)

   def talk(self, msg):
      message = self.createMessage(msg)
      print("Robot sending:", message)
      self.sock.sendto(bytes(message, 'UTF-8'), (self.address,
         self.talking_port))

   def listen(self):
      try:
         self.sock.settimeout(1/100)
         data, addr = self.sock.recvfrom(1024)
         message = data.decode('UTF-8')
         # print("received", message)
         world_dict = json.loads(message)
         return world_dict
      except socket.error:
         # print(socket.error.errno)
         pass
      return None

   def communicate(self, msg):
      self.talk(msg)
      return self.listen()


##### Robot/Controller.py #####
class Controller:

   def __init__(self, index):
      self.graphic_mode = graphic_mode
```

```
        if self.graphic_mode:

            self.view = RobotView()

        self.world = R_WorldModel()
        self.index = index

        self.message = {
            'movement_vector': Vector2(polarCoords=(0.0, 0.0), phi=0.0),
            'head_angle': 0.0
        }
        self.communicator = Communicator(self.index)

    def getWorld(self):
        return self.world

    def updateSimulation(self):
        if self.graphic_mode:
            for event in pygame.event.get():
                if event.type == QUIT:
                    print("")
                    return False
        self.createWorldFromDict(self.communicator.communicate(self.message))

        if self.graphic_mode:
            self.view.update(self.world)

        return True

    def createWorldFromDict(self, wdict):
        if wdict:
            self.world.objects_list.clear()

            for obj in wdict['objects_list']:
                new_obj =
                    R_BaseObjectModel(Vector2(polarCoords=obj['position']),
                    obj['kind'])
                self.world.objects_list.append(new_obj)
            self.head_angle = wdict['head_angle']

    def setMovementVector(self, movement_vector):
        self.message['movement_vector'] = movement_vector

    def setHeadAngle(self, head_angle):
        self.message['head_angle'] = head_angle
```