HUMBERTO CARDOSO MARCHEZI

Um Ambiente Gráfico para Desenvolvimento de Software de Controle para Robôs Móveis Através de Simulação 3D

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Engenharia Elétrica, na área de concentração em Automação. Orientador: Prof. Dr. Rer. Nat. Hans-Jorg Andreas Schneebeli.

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

VITÓRIA

2007

.

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Marchezi, Humberto Cardoso, 1978-

M317a Um ambiente gráfico para desenvolvimento de software de controle para robôs móveis através de simulação 3D / Humberto Cardoso Marchezi. - 2007.

81 f. : il.

Orientador: Hans-Jorg Andreas Schneebeli.

Dissertação (mestrado) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Robótica. 2. Simulação (Computadores digitais) 3. Sistema de computação virtual. 4. Robôs móveis. I. Schneebeli, Hans-Jorg Andreas. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 621.3

.

HUMBERTO CARDOSO MARCHEZI

Um Ambiente Gráfico para Desenvolvimento de Software de Controle para Robôs Móveis Através de Simulação 3D

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do Grau de Mestre em Engenharia Elétrica – Automação.

Aprovada em 19 de Outubro de 2007.

Comissão Examinadora:

Prof. Dr. Rer. Nat. Hans-Jorg Andreas Schneebeli
Universidade Federal do Espírito Santo, Orientador

Prof. Dr-Ing. Wu Shin Ting
Universidade Estadual de Campinas

Prof. Dr. em Engenharia Electrotécnica. Thomas Valter Rauber
Universidade Federal do Espírito Santo

Vitória, Outubro de 2007

Dedicatória

Aos meus pais, ao meu irmão, à Erika, à minha chefe Claudinete e ao professor Hans.

Agradecimentos

Primeiramente gostaria de agradecer a minha família, incluindo meus pais, João Baptista e Myrian, o meu irmão, João Paulo, e a minha noiva, Erika, por terem me apoiado durante todo esse tempo em que estive cursando o mestrado e pela paciência que tiveram comigo.

Gostaria de agradecer a minha chefe, Claudinete, por me ajudar a compatibilizar o trabalho com as tarefas acadêmicas, pois sem ajuda nunca teria sido possível terminar o curso.

Ao final e não menos importante gostaria de agradecer ao professor Hans pelo apoio que me deu desde o início do curso e por sempre estar disponível para me orientar com esta dissertação de mestrado. Suas idéias ampliaram os horizontes desse trabalho.

Humberto Cardoso Marchezi

"A imaginação é mais importante que o conhecimento."

Albert Einstein

Sumário

Lista de Tabelas

Lista de Figuras

Resumo

Este trabalho demonstra o desenvolvimento do IRCE (Integrated Robot Control Environment), um ambiente integrado para desenvolvimento de *software* de controle de uma população de robôs móveis que engloba edicão, compilacão e execucão. Ele usa as ferramentas do projeto Player/Stage/Gazebo e permite a configuracão de robôs e seus sensores além do ambiente no qual eles atuam.

O ambiente descrito permite que os algorítimos do *software* de controle possam ser desenvolvidos para depois serem verificados através de uma simulação 3D. Se desejado, o mesmo *software* de controle pode ser carregado em um robô real sem alteracões o que possibilita um ciclo mais rápido de desenvolvimento.

Além de tornar mais ágil o processo de desenvolvimento de controle para robôs móveis, o sistema também pode apoiar a pesquisa de controle inteligentes e o ensino de robótica nas universidades.

Os requisitos e a estrutura para desenvolvimento desse ambiente foram levantados usando casos-de-uso e um procedimento sistemático de desenvolvimento usando a linguagem UML (*Unified Modelling Language*) foi adotado para especificacão e documentacão do projeto. Adicionalmente, o padrão de projeto MVC (*Model-View-Controller*) foi adotado pois facilita a manutenção ou a extensão do código-fonte.

Um caso de exemplo mostra a aplicação desse ambiente para o desenvolvimento de um controle simples de desvio de obstáculos para um robô móvel. A aplicacão envolvendo vários robôs é simples de se conceber. Uma das funcionalidades mais importantes do sistema é a possibilidade de descrever uma cenário virtual de forma mais interativa com uso do mouse para alterar a posição ou as propriedades do modelos 3D envolvidos na simulação. Tal cenário é por sua vez salvo no formato de arquivo *world*, utilizado para descrever um cenário virtual 3D no projeto Player/Stage/Gazebo.

Embora existam sistemas semelhantes, uma contribuição dessa dissertacão está em apresentar um sistema de desenvolvimento integrado de código-aberto, de fácil uso e de fácil extensibilidade.

Abstract

This work demonstrates the development of an Integrated Robot Control Environment (IRCE), an integrated environment for control software development for a population of mobile robots that includes edition, compilation and execution. This environment uses the tools from the Player/Stage/Gazebo project and allows the configuration of robots and their sensors besides the environment where they actuate.

The described environment lets the control software algorithms to be developed and later verified through a 3D simulation. If desired, the same control software can be loaded in a real robot without changes what makes it possible to have a faster development cycle.

Besides turning the mobile robot software development process more agile, the system can also support the research of intelligent control and the teaching of robotics in the universities.

The requirements and the development structure of this environment were captured with use cases and a systematic development procedure using the UML language (Unified Modelling Language) was adopted to specify and document the project. Additionally the MVC design pattern (Model-View-Controller) was adopted since it eases the source-code maintenance and extension.

An example case shows how to apply this environment to develop a simple control for obstacle avoidance in a mobile robot. An application involving several robots is simple to be conceived. One of the most important functions of this system is the possibility to describe a virtual scene in more interactive manner by using the mouse to modify the position or the properties of 3D models involved in the simulation. In its turn, the scene is saved in the world file format which defines a 3D virtual scene in the Player/Stage/Gazebo project.

Although there are similar systems, one contribution of this dissertation is in present an open-source integrated development system for easy use and extensibility.

1 Introdução

Para que um robô desempenhe tarefas que envolvam interação com o ambiente externo é necessário que este seja controlado por um *software* específico implementado através de uma linguagem de programação.

O desenvolvimento desse *software* é complicado, pois a inteligência que neles é implementada é responsável por controlar o comportamento dos robôs ao interagir com obstáculos ou com outros robôs no ambiente externo.

A proposta central deste trabalho é mostrar ser possível o desenvolvimento de um ambiente gráfico integrado para desenvolvimento de *software* de controle para robôs móveis para ser utilizado nas universidades e nos centros de pesquisa, a fim de reduzir o ciclo de desenvolvimento de um *software* de simulação.

1.1 Motivação

Para permitir o desenvolvimento de aplicações mencionadas acima, vários projetos já foram propostos. Um deles é o projeto OROCOS, cuja intenção, de acordo com [1], é disponibilizar bibliotecas independentes de plataforma e implementar componentes adaptáveis e configuráveis em tempo real para a simulação e controle de sistemas robóticos. O foco está em componentes orientados a objetos e padrões de projeto. Entretanto, de acordo com [2], as desvantagens principais desse projeto são:

- *Solução complexa para problemas simples:* problemas de controle menores podem não requerer métodos de tempo-real ou avançados de controle com acesso assíncrono aos dados. Para esses casos, por exemplo, o OROCOS pode ser muito complicado e requer a absorção de muito conhecimento para se programar uma tarefa simples.
- *O alto desacoplamento resulta em vários sub-projetos:* o projeto possui várias APIs pequenas, o que significa que o código e a documentação são divididos em muitas bibliote-

cas e diretórios. Isso pode atrasar novos usuários interessados em adquirir uma idéia do projeto como um todo ou encontrar as soluções que eles estão procurando.

Outro projeto é o RoboSIM [3], cuja característica principal é o uso de um controlador real de robô para processar as tarefas de controle. Isso permite monitorar dispositivos reais ao mesmo tempo em que um robô é simulado, além de manter o comportamento temporal entre um *software* controlando um robô ou o mesmo controlando um robô simulado. As desvantagens são a necessidade de se usar um controlador real de robô e também o fato de que, para cada novo dispositivo adicionado à simulação, um conjunto de classes específicas deve existir. Isso implica que o projeto deve ser recompilado para que as mudanças tenham efeito, e obriga a um conhecimento mais profundo sobre o código-fonte do projeto, o que pode consumir muito tempo.

Há inúmeros outros projetos que também foram desenvolvidos, como o ROBOOP [4] e o QMotor [5], com abordagens semelhantes a dos projetos mencionados anteriormente e cujo foco está apenas no controle e simulação de manipuladores.

Por outro lado, muitas aplicações de robótica nas universidades e centros de pesquisa envolvem o uso de robôs com rodas e que, por isso, demandam um controle mais simples já que os graus de liberdade são menores mas, por outro lado, a interação com o ambiente externo é muito importante. Para essas aplicações, os projetos acima ainda não são adequados por tornarem demasiadamente complexo esse tipo de simulação devido a complexidade envolvida para que um novo usuário possa aprender sobre o projeto e encontrar o que precisa.

É importante notar que, de acordo com [6], atualmente, os robôs, em várias universidades no mundo, vêm equipados com uma placa TCP/IP ou *wireless 802.11* como padrão, e, hoje, o custo, a disponibilidade e a facilidade de uso desses equipamentos têm sido posto ao alcance da maioria dos usuários profissionais e acadêmicos. A comunicação com e entre os robôs no laboratório passa a ser mais barata e fácil e os ambientes de programação para robôs devem tirar proveito dessa nova realidade.

1.2 O Projeto Player / Stage / Gazebo

Uma alternativa promissora é o projeto Player / Stage / Gazebo (P/S/G) [7], inicialmente desenvolvido pela University of Southern California, que apresenta as seguintes vantagens:

- Consiste de ferramentas que se concentram no desenvolvimento rápido de aplicações de controle envolvendo uma população de robôs móveis e sensores.

- Essas ferramentas já são consideradas um padrão *de facto* na comunidade que usa código aberto em robótica [8] e há um número significativo de usuários.

O projeto consiste em três aplicativos principais: Player, Stage e Gazebo.

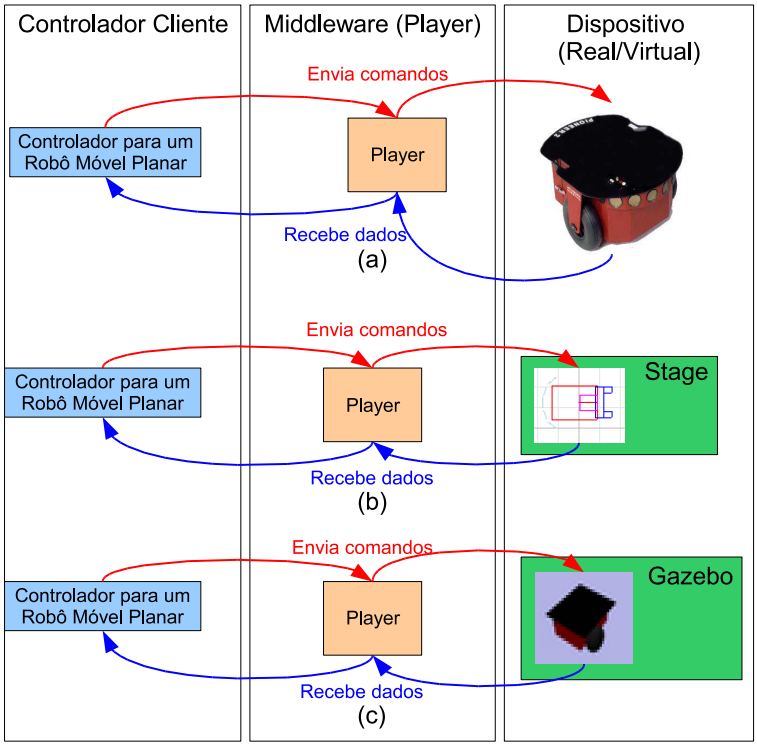


Figura 1.1: Um controlador para um modelo de robô Pioneer 2DX pode ser executado de três formas: (a) Player controla diretamente um *hardware* real. (b) Player controla um *hardware* virtual simulado pelo Stage. (c) Player controla um *hardware* virtual simulado pelo Gazebo.

O Player é um servidor de controle de robôs [6] e funciona como se fosse uma máquina virtual para dispositivos de hardware, como um modelo de robô Pioneer 2AT ou um sensor a laser SICK LMS 200, abstraindo os detalhes desses dispositivos robóticos dos controladores por meio de um canal de comunicação que pode variar desde uma rede TCP/IP quando o controle é feito a distância ou até através de processo interno de comunicação do sistema operacional quando o controle e os dispositivos robóticos estiverem no mesmo *hardware*.

Stage e Gazebo são dois simuladores que funcionam em conjunto com o Player para testar a execução de programas controladores num ambiente virtual e controlado. Eles simulam um mundo virtual enviando ao Player dados correspondentes ao funcionamento de dispositivos virtuais e recebem do Player os comandos enviados por controladores. Stage faz a simulação de uma população de robôs móveis num ambiente bidimensional. Gazebo faz o mesmo, utilizando, por sua vez, um mundo virtual tridimensional.

O *software* dos controladores pode ser implementado por um desenvolvedor ou pesquisador em C ou C++ com o uso de bibliotecas do Player, que permitem acessar o *hardware* do robô.

A integração do Player com os simuladores e os controladores é mostrada na Figura 1.1.

Ao utilizar um cenário tridimensional, a simulação de um robô passa a ter mais fidelidade por ser mais próxima da realidade, permitindo a detecção de certos problemas que não poderiam ser previstos por um simulador 2D como o Stage [9], por exemplo. Num cenário bidimensional, certos detalhes poderiam não ser representados, como no caso da mesa da Figura 1.2, que só pode ser visualizada como um retângulo ou como quatro pontos representando os seus suportes; ou no caso de superfícies irregulares ou rampas que não podem ser representadas, limitando assim a simulação no plano. Por essa razão, o Gazebo será utilizado neste trabalho.

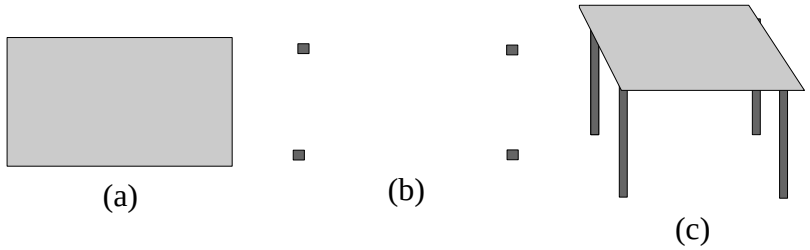


Figura 1.2: Mesa pode ter representação ambígua ao utilizar um espaço bidimensional, mas terá apenas uma no espaço tridimensional. (a) Bidimensional - cobertura da mesa. (b) Bidimensional - suportes da mesa. (c) Tridimensional.

O Gazebo é um sistema capaz de simular uma população de robôs, sensores e objetos num mundo tridimensional virtual. Para descrever esse mundo virtual, o usuário deve editar um arquivo (*.world) no formato XML, que descreve os robôs, a superfície de trabalho, o céu e os obstáculos existentes no ambiente tais como blocos, paredes e rampas dos quais o robô deve se desviar ou com os quais deve interagir.

No caso do Player, o usuário deve editar um arquivo de configuração em que ele declara quais os dispositivos serão disponibilizados para os programas controladores. Para cada dispositivo, uma *interface* é associada junto a um *driver*. Uma *interface* representa um conjunto de dispositivos de características semelhantes tais como câmeras, sonares, robôs móveis, etc. Devido a isso, para aumentar a reusabilidade do código do controlador, as bibliotecas de desenvolvimento de *software* do controlador permitem apenas o acesso a essas *interfaces*.

Num cenário simples, para utilizar o Player junto com Gazebo é preciso seguir uma seqüência de passos:

1. Editar manualmente o arquivo world (.world) para o Gazebo.

2. Editar o arquivo de configuração (.cfg) do Player.
3. Editar e compilar o código-fonte do programa controlador.
4. Executar Gazebo passando o arquivo (.world).
5. Executar o Player passando o arquivo de configuração (.cfg).
6. Executar o controlador.
7. Observação da simulação e repetição desses passos, se necessário.

1.3 Definição do Problema

Conforme mostra a Figura 1.3, para uma simples simulação são necessários três processos (um para executar o Gazebo, outro para executar o Player e um outro para compilar e executar o *software* do controlador), e três processadores de texto (um para editar o arquivo do mundo virtual no Gazebo, outro para editar o arquivo de configuração no Player e outro para editar o código-fonte do *software* do controlador). Para cada robô adicionado na simulação, um processador de texto deve ser aberto para editar o código do *software* do controlador, outro processador de texto para editar o arquivo de configuração, um processo para executar o arquivo de configuração no Player e finalmente outro processo para para compilar o controlador e executar o mesmo após compilado. Ou seja, são dois processadores de texto e dois processos para cada robô adicionado à simulação.

Logo, dois problemas principais podem ser identificados:

1. Os processos Player e Gazebo não estão integrados num mesmo ambiente, requerendo chaveamento de contextos numa simples simulação seguida de controle de robôs.
2. O projeto Player/Stage/Gazebo não oferece uma ferramenta para modelagem de cenas de forma interativa.

Para resolver esses problemas, este trabalho propõe a construção de um ambiente integrado de desenvolvimento de controles para robôs móveis, abreviado como **IRCE** (Integrated Robot Control Environment) [10], que deve atingir os objetivos:

- **Aumento da Agilidade do Processo de Desenvolvimento de Controle para Robôs Móveis**

O ambiente deve ser capaz de integrar as ferramentas Player e Gazebo de forma transparente.

- **Apoio à Pesquisa de Controladores Inteligentes**

  Fazer com que o pesquisador ou programador se concentre mais na tarefa de desenvolver *softwares* mais inteligentes de controladores.

- **Apoio ao Aprendizado de Robótica**

  O ambiente deve ser de utilidade para alunos de universidades ou programadores interessados em aprender de maneira concreta sobre o desenvolvimento de aplicações de robótica.

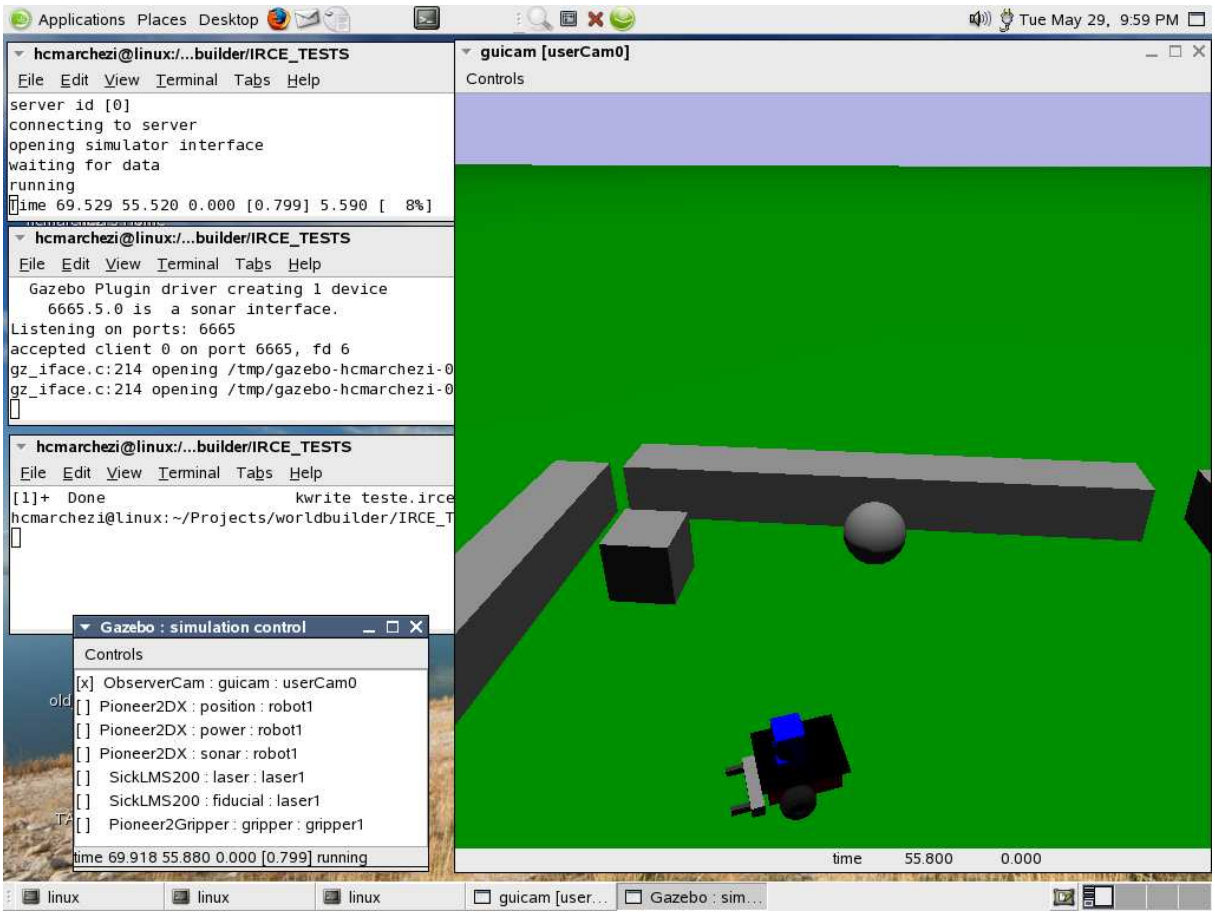


Figura 1.3: Vários processos são necessários para executar uma simulação com único robô.

1.4 Metodologia de Trabalho

O trabalho realizado nessa dissertação caracteriza-se por ser uma pesquisa aplicada, exploratória e experimental [11] e dividi-se pelas seguintes etapas:

1. **Familiarização com as Ferramentas do Projeto Player/Stage/Gazebo**

   O desenvolvimento dessa ferramenta começa por um período de intensas pesquisas para entender o funcionamento das ferramentas do projeto P/S/G tanto pelo ponto-de-vista do usuário quanto do ponto-de-vista da implementação interna do projeto.

   A ferramenta Gazebo do projeto P/S/G é implementada utilizando a linguagem de programação C++ e as bibliotecas utilizadas são:

   - *OpenGL*: para visualização de objetos tridimensionais [12].
   - *ODE*: para simulação de comportamento de eventos físicos [13].
   - *wxWidgets*: para a implementação da interface gráfica [14].
   - *libxml2*: biblioteca em C para manipulação de arquivos XML [15].

2. **Escolha do Ambiente de Desenvolvimento do Trabalho de Dissertação**

   Como a reusabilidade de código é desejável para facilitar o trabalho de implementação do ambiente de desenvolvimento, a mesma linguagem de programação e bibliotecas é utilizada na implementação do protótipo desse trabalho. A biblioteca *libxml2* é muito popular, mas por ser difícil de utilizar, a biblioteca *TinyXML* [16] é utilizada para a geração de arquivos XML.

3. **Escolha da Forma de Documentação do Protótipo**

   Seguindo o padrão adotado na implementação do Gazebo e para abstrair melhor a complexidade do problema, o paradigma orientado a objetos é usado. Por isso, o padrão UML [17] é adotado para a documentação do trabalho proposto.

4. **Análise e Projeto Orientado a Objeto do Protótipo**

   Fazendo uso da forma de documentação escolhida, o processo de desenvolvimento do protótipo é iniciado seguindo as etapas de análise e projeto orientados a objeto.

5. **Projeto de *Interface***

   Seguindo a modelagem produzida na fase de projeto, as interfaces gráficas do sistema são desenhadas até que os objetivos sejam atingidos da melhor maneira possível.

6. **Implementação de um Protótipo**

   O primeiro passo é a construção um protótipo que permitisse a visualização do mundo virtual descrito no arquivo formato world do Gazebo que permita que o usuário manipule objetos gráficos [18], no caso, os modelos, editando as suas propriedades e arrastando no mundo virtual para alterar a sua posição no espaço. Os modelos 3D são extensões de classes em C++ reutilizadas do Gazebo para serem usados no IRCE.

   Após essa fase, o restante das funcionalidades requeridas pelo sistema, que exigem menos trabalho, são implementadas até chegar a um protótipo utilizável.

7. **Análise dos Resultados**

   O trabalho de análise de resultados inicia-se quando o protótipo chega a uma configuração estável. Essa fase é responsável por testar e verificar se os objetivos do sistema foram alcançados pelas funcionalidades do sistema.

1.5 Ambientes Gráficos de Desenvolvimento

Com o intuito de facilitar o desenvolvimento de aplicações com robôs, alguns projetos envolvendo extensões das ferramentas do Projeto P/S/G foram desenvolvidas tais como:

- **Pyro** [19] [20]: um ambiente de programação que permite ao desenvolvedor explorar tópicos em inteligência artificial e robótica sem se preocupar com detalhes internos do hardware controlado. Pyro suporta robôs da família *Pioneer*, *Khepera*, *AIBO*, *IntelliBrain-Bot* e *Roomba* e funciona como uma camada que controla os simuladores *Robocup soccer*, Player/Stage, Gazebo e o simulador do *Khepera*. Embora permita a edição interativa do mundo virtual 2D usado pelo Stage, não possui a mesma funcionalidade de edição interativa para o Gazebo. Ou seja, o desenvolvedor ainda tem que editar o arquivo do cenário 3D manualmente ao utilizar o Gazebo.

- **Eclipse Robot IDE Plugin** [21]: plugin desenvolvido para o ambiente de programação Eclipse [22] que oferece uma opção integrada de se trabalhar com o desenvolvimento de robôs envolvendo Player e Stage. Ao utilizar o Eclipse, o projeto permite que o desenvolvedor tire proveito de funcionalidades ja desenvolvidas para a implementação e depuração do código dos controladores. Por outro lado, possui suporte apenas para o Stage não sendo possível utilizar esse plugin para simulações que envolvam cenas mais realísticas em 3D.

Fora do Projeto P/S/G, o sistema que mais se destaca é o Webots. Webots é um *software* comercial usado para simulação de protótipos de robôs móveis e transferência de controle para robôs reais [23] [24]. Esse *software* permite a edição interativa do mundo virtual e oferece simulação realística da física e permite ao usuário programar robôs nas linguagens C, C++ e Java via TCP/IP, e transferir o controle para robôs móveis reais. No entanto, o Webots é um pacote comercial fechado, o que impede que pesquisadores possam adequar o sistema às suas necessidades, além do preço da licença ainda ser proibitivo para a maioria das pessoas.

1.6 Estrutura do Trabalho

A primeira parte da dissertação apresenta os seguintes pontos:

1. Motivação para um ambiente de desenvolvimento rápido e simples para aplicações robóticas.
2. Justificativa pela utilização do projeto P/S/G como plataforma de desenvolvimento.
3. Definição do problema, que consiste em um ambiente integrado de desenvolvimento utilizando o projeto P/S/G.
4. Trabalhos relacionados ao mesmo tema.
5. Estrutura do trabalho.

A seguir, o Capítulo 2 explica em mais detalhes como é a utilização das ferramentas Player e Gazebo do projeto P/S/G e como é o desenvolvimento de um programa controlador utilizando as bibliotecas do Player.

O Capítulo 3, detalha como ocorre o processo de desenvolvimento do ambiente de desenvolvimento para aplicações com robôs. É onde são descritos os requisitos, os casos-de-uso, as classes e a arquitetura utilizada para a implementação do sistema.

Tomando como base esses requisitos, o Capítulo 4 mostra como essas funcionalidades foram projetadas no sistema, apresentando as telas do sistema e como elas funcionam.

Para dar uma idéia sobre como o ambiente funciona numa situação real, um exemplo com um cenário de uso simples é explicado passo-a-passo no Capítulo 5.

Ao final o Capítulo 6, conclui o resultado do trabalho e aponta alguns trabalhos futuros.

2 *Player e Gazebo*

2.1 Arquitetura

A Figura 2.1 mostra a arquitetura do projeto, baseada na arquitetura cliente/servidor. Os aplicativos Gazebo e Player e os controladores podem estar instalados em computadores separados desde que estejam numa mesma rede TCP/IP.

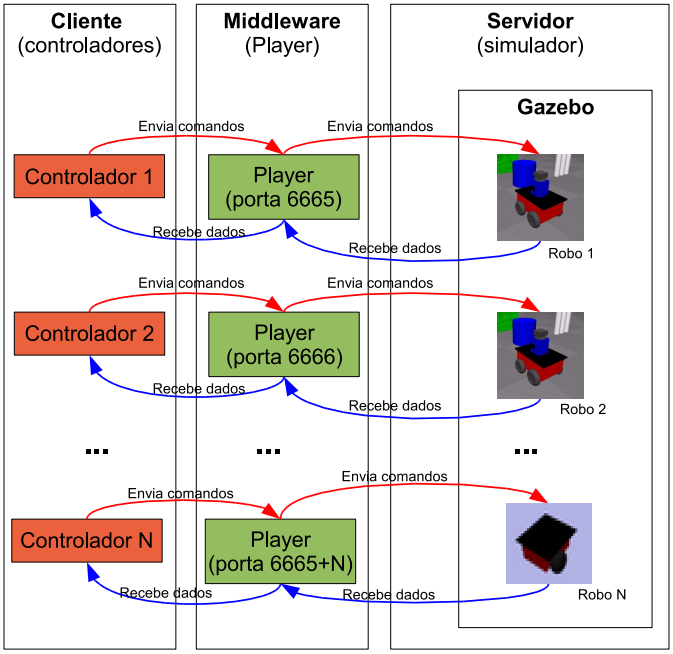


Figura 2.1: Player, Gazebo e os controladores seguem uma arquitetura cliente/servidor com o Player como *middleware*.

O aplicativo Gazebo, que simula o funciona dos dispositivos virtuais, pode ser executado num computador e pode receber comandos e enviar dados para vários aplicativos Player. Cada Player deve receber comandos e receber dados de um controlador. Esse processo é ilustrado pela Figura 2.2.

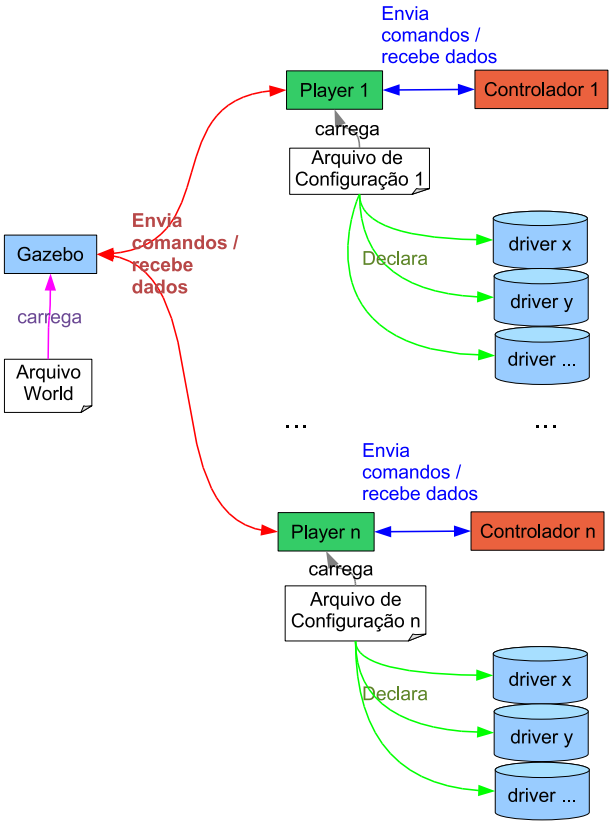


Figura 2.2: Gazebo e Player se integram para produzir uma simulação.

Para executar uma simulação, as ferramentas Gazebo, Player e o controlador são executados em processos independentes e trocam mensagens entre si. O Gazebo é configurado pelo arquivo *world* que descreve o mundo virtual e o Player é configurado por um arquivo *cfg* que descreve o *hardware* a ser controlado seja ele virtual ou real.

2.2 Player

2.2.1 Conceitos Básicos

Existem três conceitos principais: a *interface*, o *driver* e o dispositivo (*device*).

- ***Interface***

  Abtrai detalhes tecnológicos específicos de uma determinada classe de atuadores, sensores robóticos ou algoritmos em um conjunto de comandos que são usados pelos controladores. Ela define a sintaxe e a semântica das mensagens que podem ser trocadas com as *entidades* que pertencem à mesma classe. Por exemplo, a *interface position2d* define um robô móvel com rodas e permite que o programador defina a posição e velocidade desse robô, assim como receber informações do seu estado, no caso, a posição e velocidade.

  Dessa forma, a *interface position2d* é definida na classe *Position2dProxy*, que por sua vez contém os métodos *SetSpeed* (Atribui Velocidade), *Goto* (Atribui Posição), *GetPose* (Obtém Posição e Orientação), *SetMotorEnable* (Habilita ou não o motor), etc. Logo, todo robô móvel deve implementar esta *interface*.

  Pode-se ver uma lista com algumas das *interfaces* disponibilizadas pelo Player no Apêndice A.

- ***Driver***

  Funciona como um conversor das funções de "alto nível"entre a sua respectiva *interface* em comandos específicos para o funcionamento de um dado dispositivo físico ou virtual. Um dispositivo virtual emula o comportamento de um dispositivo físico mas existe apenas dentro de uma simulação no Gazebo, por exemplo. Um *driver* se comunica com um sensor, atuador ou algoritmo (como para imitar o funcionamento de equipamento de *GPS*) e traduz as entradas e saídas para obedecer a uma ou mais *interfaces*. A função do *driver* é a de abstrair detalhes específicos de uma dada *entidade* fazendo com que ela se pareça com qualquer outra entidade da mesma classe. Já existem *drivers* para uma grande variedade de *hardware*, por exemplo, o driver p2os, que pode ser usado para controlar vários robôs do fabricante ActivMedia como a série *Pioneer* e o *AmigaBot*. Por outro lado, para controlar as câmeras Sony EVI-D30 e Sony EVI-D100, o driver sonyevid30, pode ser utilizado. Além disso, um *driver* pode implementar mais de uma *interface*, como é o caso do *driver p2os*.

  Para ver alguns exemplos de *drivers* implementados para o Player, veja o Apêndice B.

- **Dispositivo (Device)**

  O dispositivo pode ser definido como um *driver* ligado a uma *interface* com um endereço único. Todas as mensagens entre os dispositivos no Player ocorrem via *interfaces*. Dessa forma, o *driver* nunca é acessado diretamente pelos controladores.

  A *interface laser*, por exemplo, define um formato de retorno de leitura de um sensor laser planar de proximidade. Por sua vez, o driver *sicklms200* controla o SICK LMS 200, um tipo específico de sensor laser planar de proximidade muito comum em robôs móveis. O *driver sicklms200* sabe se comunicar com o SICK LMS 200 usando uma porta serial e gravar dados de leitura nessa porta. Como o que se quer é a reusabilidade no programa de controle, para não acessar dados num formato específico do SICK, o *driver* também traduz os dados que vêm do SICK LMS 200 para o formato definido na *interface laser*. Através de uma *interface* implementada por um *driver* pode-se controlar o dispositivo *laser* que pode ser localizado pelo endereço: localhost:6665:laser:0 . Onde *localhost* indica o servidor em rede onde o *laser* esta, 6665 é a porta por onde se comunicar com esse servidor, o *laser* é a interface a ser utilizada e 0 indica que é o primeiro dispositivo *laser* da lista de dispositivos.

2.2.2 Arquivo de Configuração

Os dispositivos a serem controlados são declarados num arquivo de configuração que é lido pelo Player para que ele possa se comunicar com esses dispositivos. O conteúdo de um arquivo de configuração é um arquivo no formato de texto que contém uma ou mais declarações para dispositivos. Essas declarações são definidas nas seções *driver* seguidas por parênteses. Entre os parênteses estão as opções de configuração para o dispositivo. Basicamente, a seção *driver* contém as seguintes opções mostradas na Figura 2.3.

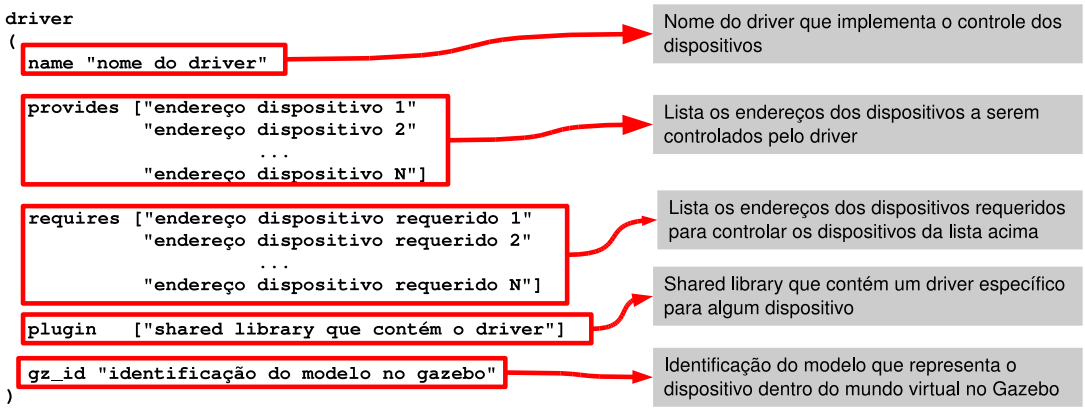


Figura 2.3: Seção driver e suas opções básicas

A Figura 2.4 mostra como declarar um modelo de robô Pioneer no arquivo de configuração através de uma seção

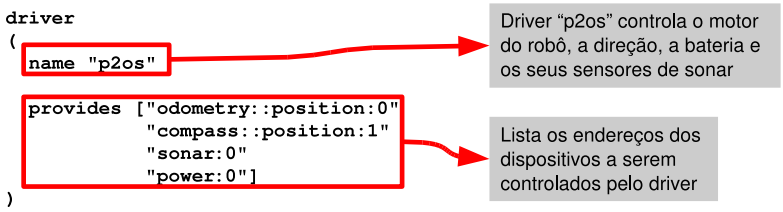


Figura 2.4: Exemplo de uma seção driver para um robô Pioneer

A configuração acima serve apenas para controlar um Pioneer real diretamente, entretanto, para usar um Pioneer numa simulação com o Gazebo, o arquivo de configuração deve conter uma seção *driver* para a simulação e outras seções *driver* para cada dispositivo do robô envolvido na simulação, conforme mostrado na Figura 2.5:

A primeira seção serve para declarar o Gazebo como um dispositivo de simulação que irá receber e enviar mensagens ao Player para simular o comportamento de dispositivos virtuais. As outras seções declaram cada dispositivo virtual a ser disponibilizado para o programa controlador.

A opção *name*, que é o nome do *driver*, é descrito sempre como Gazebo, tal como o simulador, e além disso, também deve ser declarada a opção *gz_id* que é o identificador do modelo virtual a ser controlado no arquivo do mundo virtual a ser lido pelo Gazebo.

2.2.3 Desenvolvimento do Software de Controle

Para que os desenvolvedores possam escrever aplicações que controlem robôs, a instalação do Player inclui bibliotecas que permitem se comunicar com os dispositivos declarados no arquivo de configuração. No momento, a instalação do Player contém as linguagens C, C++, Tcl e Java, mas versões para outras linguagens estão em andamento. Na verdade, como toda a comunicação entre o player e os programas de controles segue um protocolo aberto, versões para qualquer linguagem de programação podem ser desenvolvidas desde que sigam esse protocolo.

Um exemplo de programa de controlador pode ser visto no Apêndice C.

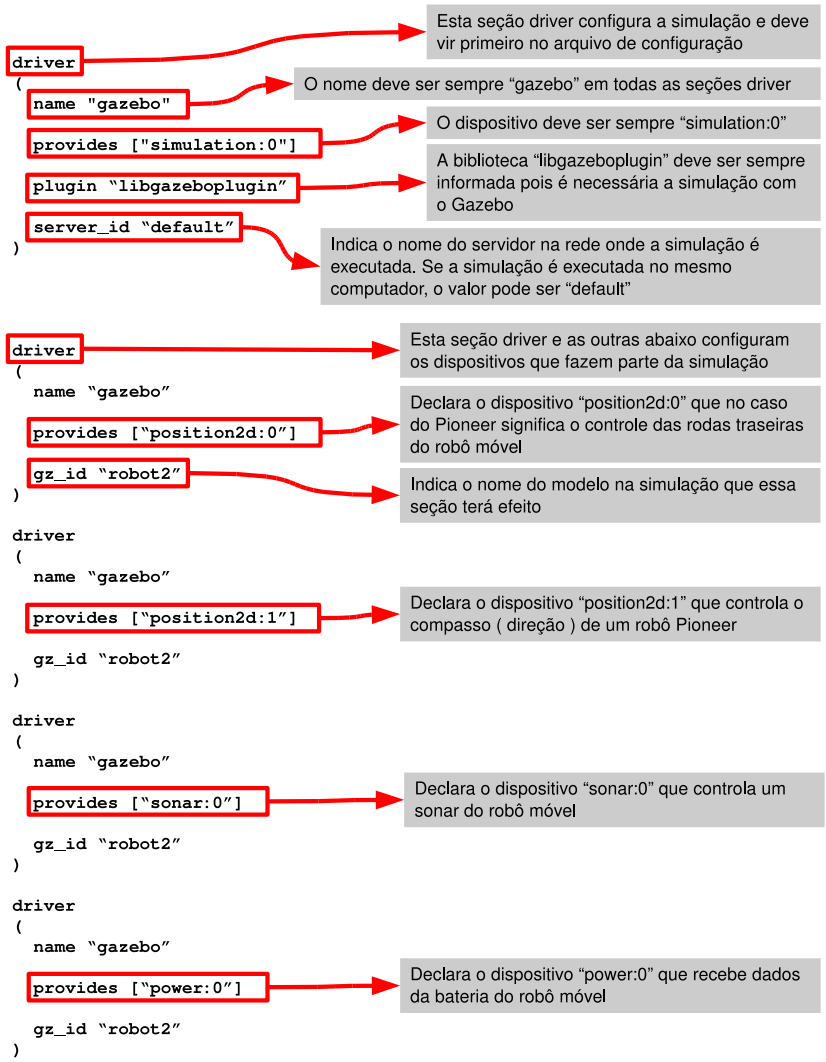


Figura 2.5: Exemplo de configuração de dispositivos para simulação utilizando um robô Pioneer

2.3 Gazebo

Gazebo é um simulador tridimensional que foi feito para interagir com o Player a fim de simular o controle em vários tipos de *hardware* de robô.

Para utilizar o Gazebo, primeiramente, o desenvolvedor deve escrever manualmente um arquivo XML específico (arquivo world) que descreve o mundo virtual.

Esse arquivo contém todos os modelos que fazem parte da simulação. O arquivo inicia-se com a *tag* padrão (*<?xml version="1.0"?>*) de um arquivo XML seguida pela *tag* do mundo virtual (*gz:world*). Entre as tags do mundo virtual, outras tags podem ser declaradas para cada modelo que faz parte da simulação. Cada tipo de modelo, possui uma etiqueta que o identifica, além disso, cada modelo deve ter um identificador que possa ser usado mais tarde pelo programa

de controle e pelo arquivo de configuração. Um modelo pode ser composto por outros modelos também. Um exemplo é mostrado no Apêndice D.

2.4 Vantagens e Desvantagens no uso do Player e Gazebo

O *software* do projeto Player roda no Linux, Solaris, *BSD e Mac OS X (Darwin) e consiste basicamente dos aplicativos Player, Stage e Gazebo.

O projeto P/S/G atinge o seu objetivo ao permitir que um desenvolvedor implemente rapidamente um controlador para um robô móvel com sensores. Os *drivers* ja incluídos com o Player oferecem acesso aos modelos mais comuns de *hardware* de robô como os modelos de robô da série Pioneer, sensores a *laser* como SICK LMS 200 e a câmera Sony VID 30.

Entretanto, além das ferramentas Player e Gazebo não estarem integradas em um ambiente de desenvolvimento de controladores e do Gazebo não suportar a edição interativa do mundo virtual, outras dificuldades foram observadas:

Uma das dificuldades principais é a edição manual de arquivos de texto.

Para fazer uma simulação no Gazebo, por exemplo, o usuário deve escrever manualmente um arquivo (*.world) no formato XML que descreve a posição, orientação e os modelos de robôs que serão usados e a descrição do mundo virtual em 3D incluindo os obstáculos. Esse processo consome tempo além de não ser prático para simulações um pouco maiores. A Figura 2.6 mostra o *workflow* para edição um arquivo (*.world).

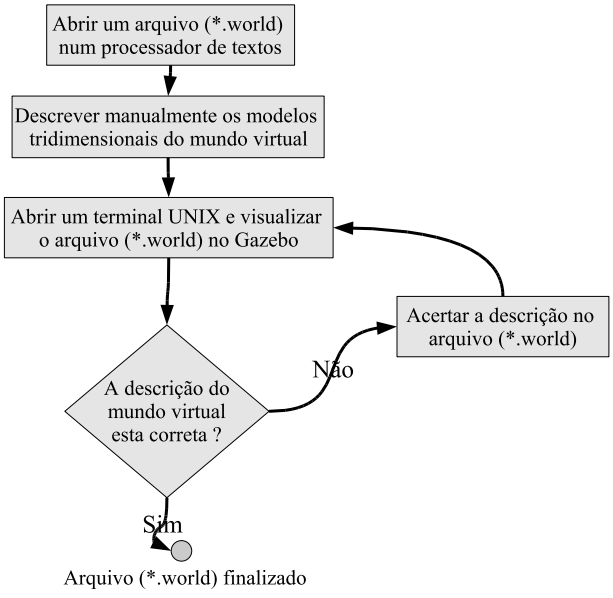


Figura 2.6: *Workflow* de montagem do mundo virtual.

No caso do Player, o usuário deve editar manualmente um arquivo de configuração. Nessa tarefa, o desenvolvedor pode facilmente errar na descrição de uma seção *driver* e pode não ser tão simples achar onde está o erro conforme ilustrado no *workflow* da Figura 2.7.

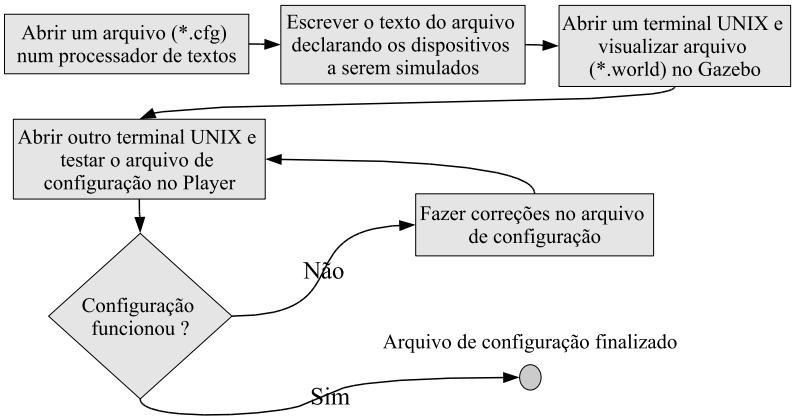


Figura 2.7: *Workflow* de configuração do robô e seus sensores.

3 *Projeto do Ambiente Gráfico de Desenvolvimento*

Este capítulo descreve as fases do processo de desenvolvimento para a construção do ambiente proposto que são:

- Captura de Requisitos
- Descrição dos Atores
- Descrição dos Casos-de-Uso
- Modelo de Classes de Análise
- Desenho da Arquitetura do Sistema

3.1 Requisitos

Para que o ambiente de desenvolvimento integrado proposto atinja os seus objetivos listados na seção 1.3, ele deve preencher alguns requisitos:

1. **Gerência dos Arquivos de um Projeto de Simulação**

   O desenvolvimento e a simulação de *software* para controladores no projeto P/S/G envolve o uso de diferentes tipos de arquivos, tais como o arquivo *world*, os arquivos de configuração e os arquivos que contêm o código-fonte dos controladores. Logo, é necessário que o sistema possua uma referência sobre quais desses arquivos fazem parte do projeto de controladores que se quer desenvolver.

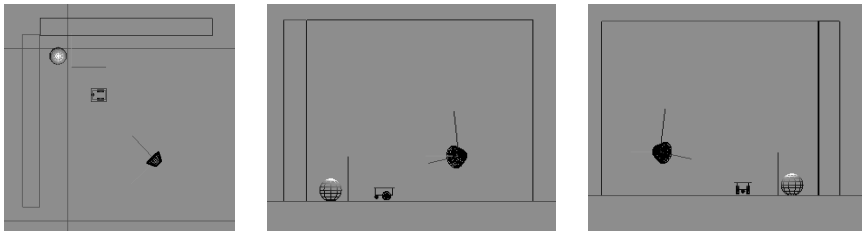
2. **Montagem Interativa do Mundo Virtual**

   Um dos objetivos listados na Seção 1.3 é de que o ambiente IRCE deve permitir a visualização do mundo virtual descrito em arquivo no formato *world* do Gazebo assim como

mover a posição dos modelos no mundo virtual e adicionar ou remover modelos do cenário a ser construído.

3. **Visualização Gráfica do Mundo Virtual em Diferentes Perspectivas**

   Durante a visualização, o desenvolvedor deve ser capaz de enxergar o mundo virtual em pelo menos três perspectivas, já que se trata de uma representação tridimensional. Para isso, foram definidas as formas ortogonais X/Y, X/Z e Y/Z, que cobrem todo o espaço tridimensional, conforme pode ser visto na Figura 3.1. O desenvolvedor deve ser capaz de selecionar uma dessas formas ortogonais numa lista de formas de visualização para verificar a descrição do mundo virtual.



(a) (b) (c)

Figura 3.1: (a) X/Y. (b) X/Z. (c) Y/Z.

Quando uma câmera virtual é adicionada ao mundo virtual, seja um observador ou um dispositivo virtual de um robô, a visualização provida por ela deve constar na lista de formas de visualização. Para seguir um padrão mais realista, ao contrário das formas ortogonais, a imagem captada pelas câmeras deve ser representada na forma de uma projeção. A visualização das câmera é mostrada na Figura 3.2.

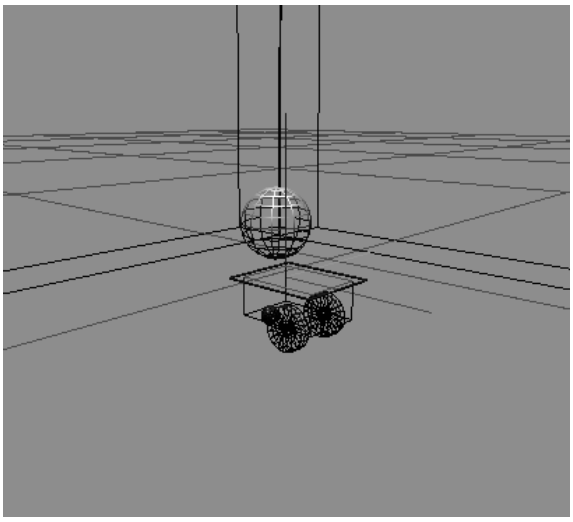


Figura 3.2: Câmera provê visualização em perspectiva.

4. **Alteração das Propriedades de um Modelo**

   Cada modelo deve possuir propriedades que descrevem estados e comportamentos do modelo durante a simulação. Para que o usuário possa, de forma interativa, alterar os modelos que fazem parte do mundo virtual, é necessário que o mesmo possa selecionar, visualizar e alterar as suas propriedades num painel gráfico em separado.

5. **Configuração Facilitada dos Robôs e seus Sensores**

   O sistema deve permitir que o desenvolvedor de controladores faça a configuração de cada robô e seus sensores envolvidos no projeto de forma facilitada para evitar de ter que se lembrar dos dados necessários para criar os arquivos de configuração.

   Para tornar a configuração mais intuitiva, ao invés de ter que se lembrar do formato do arquivo de configuração e editá-lo manualmente, o desenvolvedor deve apenas preencher campos na *interface* gráfica estabelecidos, usados pelo sistema para gerar automaticamente o arquivo de configuração, após o salvamento, conforme mostrado na Figura 3.3. Ao carregar um arquivo de configuração, o sistema deve extrair os dados e carregá-los em seus respectivos campos da *interface* gráfica.

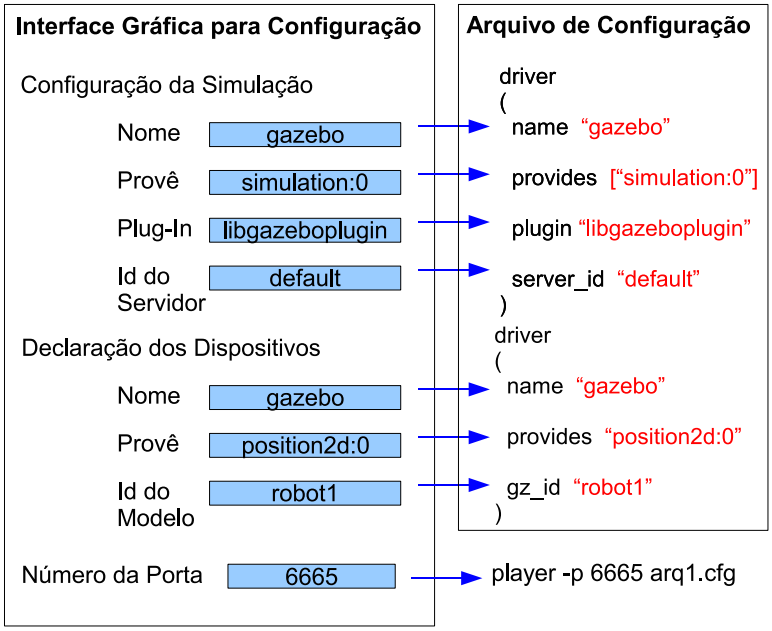


Figura 3.3: Interface gráfica para preencher o arquivo de configuração.

6. **Desenvolvimento Integrado do Software dos Controladores**

   O sistema deve permitir que um desenvolvedor tenha a habilidade de criar, editar e excluir os arquivos fonte dos controladores. Além disso, o desenvolvedor deve ser capaz de compilar esses arquivos produzindo o controlador e de visualizar possíveis erros de implementação durante o processo de compilação caso esses existam.

7. **Execução Simplificada da Simulação**

   Após descrever o mundo virtual com os modelos, os arquivos para configurar os dispositivos usados e implementar o *software* dos controladores, o ambiente deve automatizar as tarefas necessárias para executar uma simulação e permitir a verificação do funcionamento dos controladores pelo desenvolvedor.

3.2 Casos de Uso

Seguindo o padrão UML, os requisitos do sistema são traduzidos para o formato de casos-de-uso. Os casos-de-uso descrevem o comportamento do sistema sob várias condições à medida que este responde a uma requisição de um dos usuários-chave, chamados de atores primários; o ator primário inicia a interação com o sistema para atingir um objetivo; o sistema responde, protegendo o interesse de todos os usuários-chave [25].

3.2.1 Atores

Neste sistema pode ser identificado apenas um papel distinto desempenhado pelos usuários que é o de *Programador de Robôs*, na Figura 3.4, responsável por todas as tarefas necessárias a uma simulação com robôs móveis.

Figura 3.4: Ator - Programador de Robôs

3.2.2 Casos-de-Uso Principais

A partir dos requisitos acima, os casos-de-uso mostrados na Figura 3.5 foram capturados e são: gerência de projeto IRCE, montagem do mundo virtual, construção de robô e seus sensores, programação de controle e simulação.

A seguir os casos-de-uso acima são detalhados e alguns casos-de-uso principais foram expandidos em outros casos-de-uso menores.

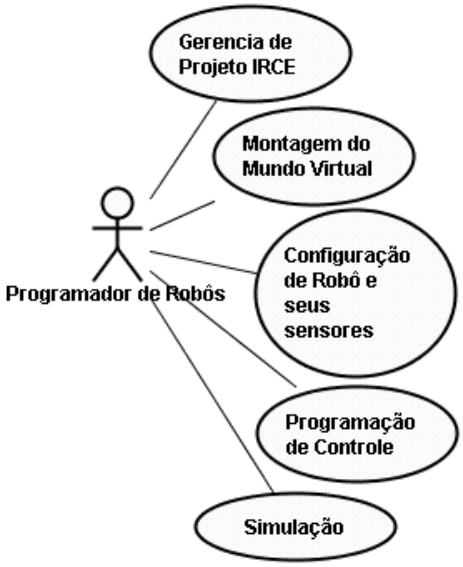


Figura 3.5: Casos de uso principais da ferramenta IRCE.

3.2.3 Gerência de Projeto

O caso-de-uso Gerência de Projeto é expandido em outros casos-de-uso, mostrados na Figura 3.6, e tratam justamente das funcionalidades de criar, editar e salvar o arquivo de projeto e preenche o requisito 1.

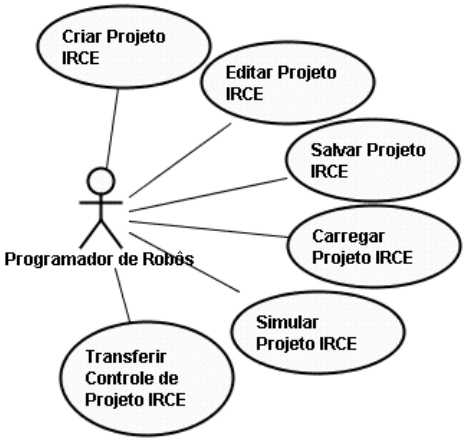


Figura 3.6: Cenários do caso-de-uso de gerência de projeto.

Para atingir o objetivo desse caso-de-uso, as classes que representassem o projeto com o mundo virtual, as configurações e os programas de controle foram criadas conforme o diagrama UML da Figura 3.7 foram criadas para tratar diretamente dessa funcionalidades.

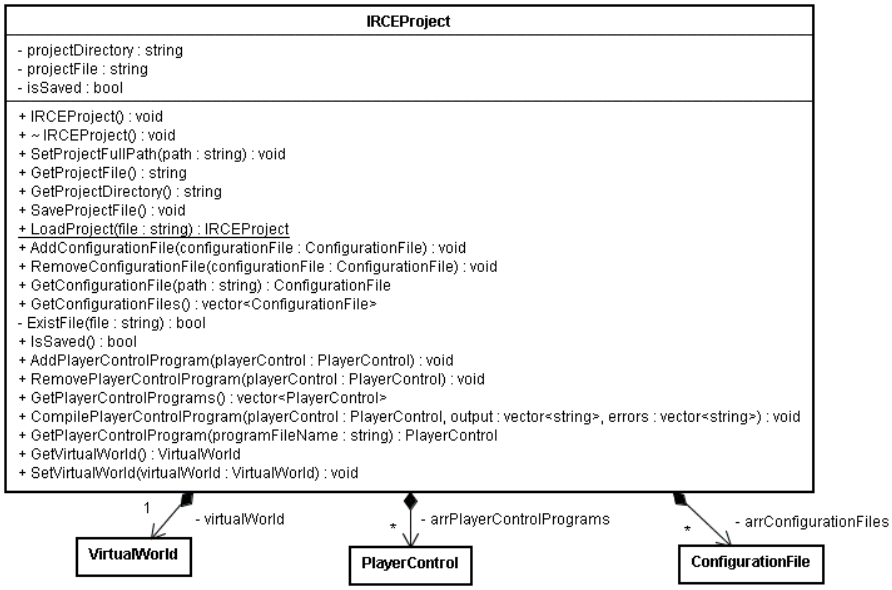


Figura 3.7: Diagrama de classes de gerência de projeto.

3.2.4 Montagem do Mundo Virtual

Este caso-de-uso trata das cenários de criar, carregar e remover um arquivo *world* no projeto e também de adicionar, editar e remover modelos tridimensionais no mundo virtual descrito pelo arquivo *world*. A edição do mundo virtual ocorre ocorre sempre no modo *WYSIWIG* (o que você vê é o que você tem).

A Figura 3.8 ilustra os de casos-de-uso para a montagem do mundo virtual que preenchem os requisitos 2, 3 e 4.

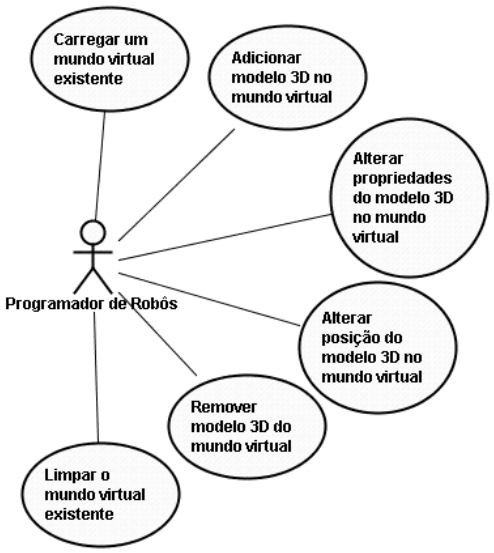


Figura 3.8: Cenários do caso-de-uso de montagem do mundo virtual.

Cada elemento visual que compõe o mundo virtual é um modelo tridimensional que deve saber se desenhar no espaço e é representado por uma classe reutilizada diretamente do código do Gazebo. Como cada modelo tem como característica comum saber se desenhar, estes devem herdar da classe *Model* conforme mostrado na hierarquia da Figura 3.9 que mostra alguns dos modelos adaptados para esse ambiente de programação.

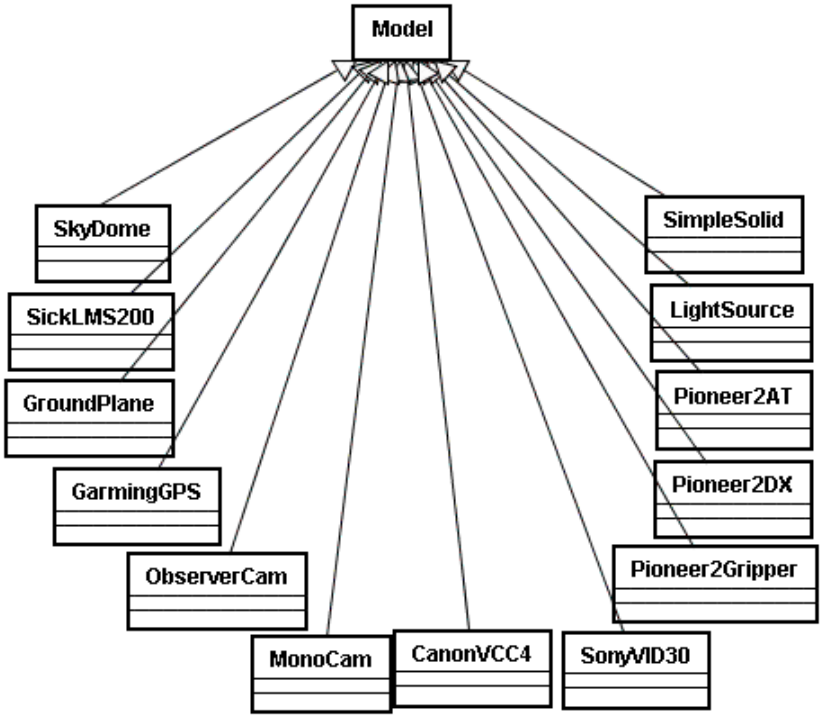


Figura 3.9: Hierarquia de classes de modelos usados na simulação.

Após a visualização, o requisito 4 diz respeito à seleção de modelos pelo desenvolvedor e à visualização de suas propriedades dinamicamente em uma barra de propriedades que são extraídas de um modelo qualquer em tempo de execução.

Para permitir que novos modelos possam ser mais facilmente usados, o sistema não deve saber qual o modelo específico esta sendo usado no momento mas deve saber extrair todas as propriedades por meio de polimorfismo utilizando apenas um modelo genérico. Como as classes em C++ não possuem essa capacidade, uma infra-estrutura teve de ser criada para tornar isso possível. A classe *Model*, que representa um modelo genérico no Gazebo, deve ter o seu código e deve herdar da classe *FieldDiccionary*, que contém uma estrutura de dados no formato de um mapa que serve como um dicionário de propriedades e que pode retornar o valor de uma propriedade pelo nome, conforme mostrado na Figura 3.10.

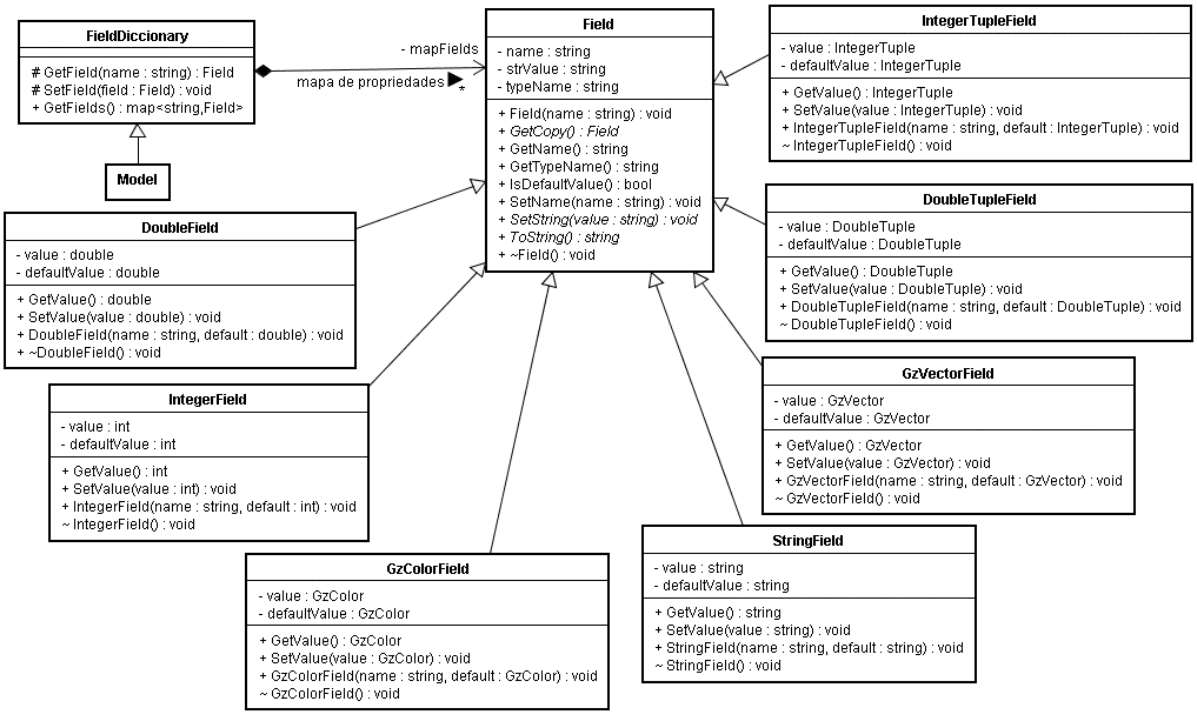


Figura 3.10: Diagrama de classes mostra o mapa de propriedades e a hierarquia de propriedades.

Essas propriedades também são utilizadas pelo sistema para serializar os modelos do mundo virtual para um arquivo XML ou o caminho inverso instanciando os modelos do arquivo *world* para atingir parte do requisito 1.

Esse processo é realizado pelo classe *XMLCommand* na Figura 3.11.

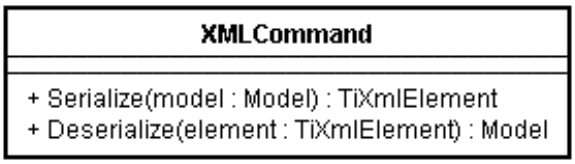


Figura 3.11: Classe responsável por serializar os modelos.

3.2.5 Configuração de Robô e seus Sensores

As funcionalidades desse caso-de-uso preenchem o requisito 5 de criação, edição, remoção e salvamento de arquivos de configuração pelo desenvolvedor, e os seus casos-de-uso são mostrados na Figura 3.12.

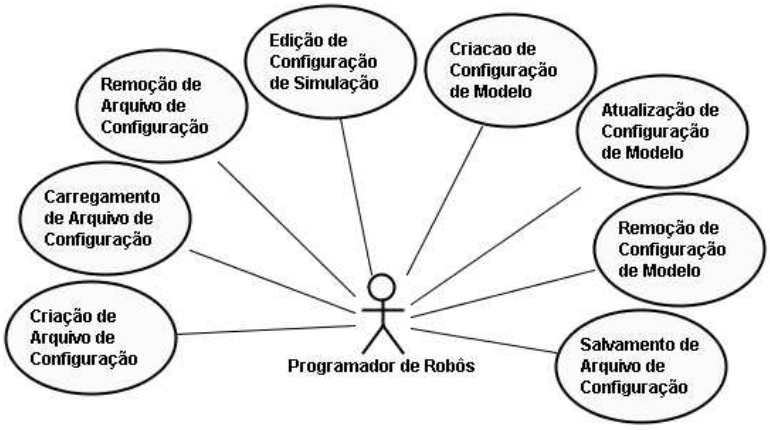


Figura 3.12: Cenários do caso-de-uso de configuração de robô e seus sensores.

Para tornar o caso-de-uso possível, a estrutura de classe da Figura 3.13 é criada. A classe *ConfigurationFile* representa um arquivo de configuração utilizado pelo Player e é composto por seções driver para configuração conforme mostrado na Figura 2.2.2. Um arquivo de configuração é composto por uma configuração de simulação (*SimulationConfiguration*) e por uma ou várias configurações de dispositivo (*PlayerConfiguration*). Tais elemento são representados pelo diagrama da Figura 3.13.

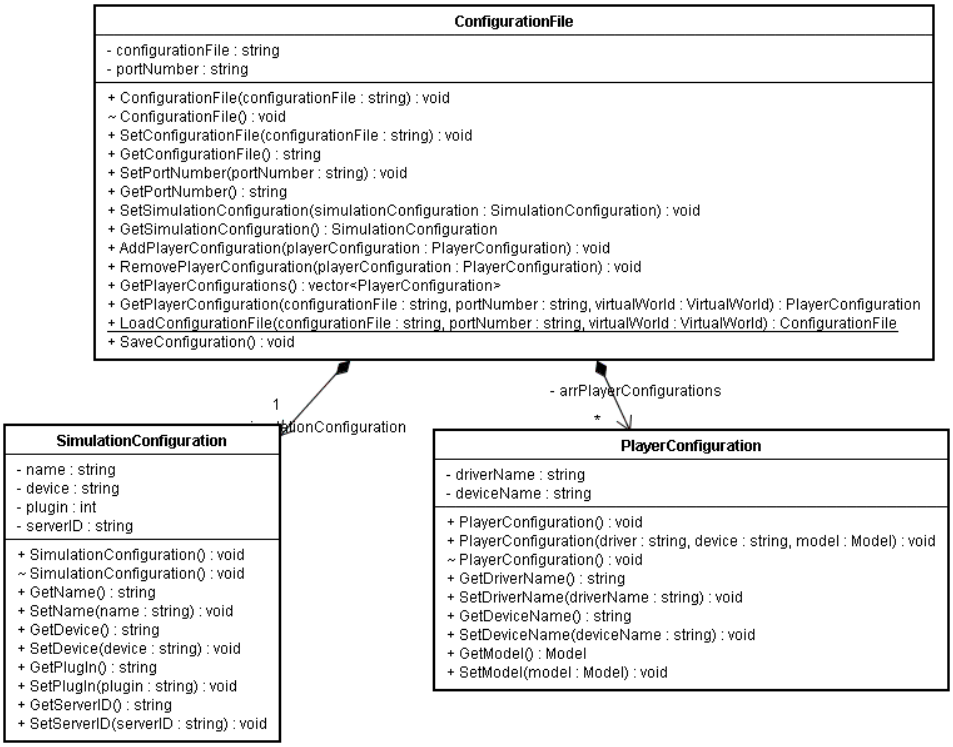


Figura 3.13: Classes de Configuração de Robôs.

3.2.6 Programação de *Software* de Controlador

A Figura 3.14 representa os casos-de-uso necessários para atingir o requisito 6 de desenvolvimento de *software* de controlador.

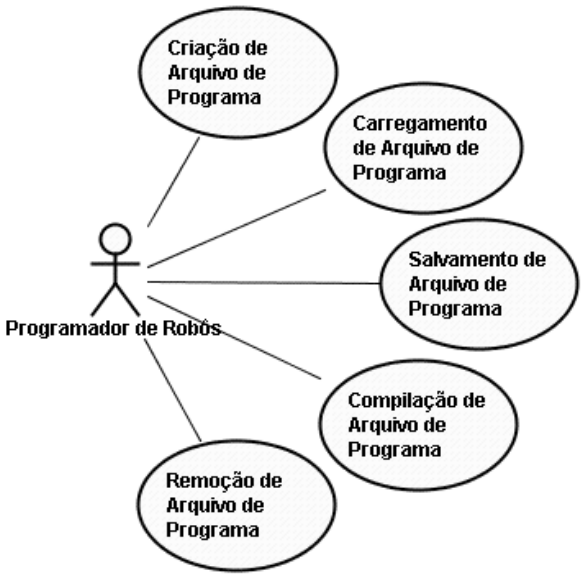


Figura 3.14: Cenários do caso-de-uso de programação de *software* de controle.

O arquivo do código-fonte do controlador é representado pela classe PlayerControl, representado na Figura 3.15. A classe PlayerControl contém o texto do código-fonte, a linguagem de programação e o nome do arquivo onde deve ser armazenado.

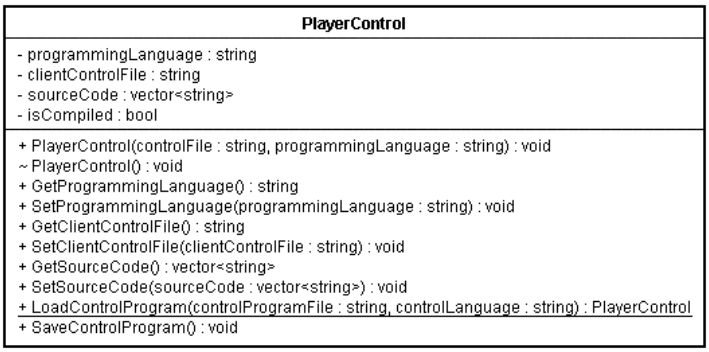


Figura 3.15: Classe que representa o programa controlador.

3.2.7 Simulação

Conforme o requisito 7, a simulação passa a ser automatizada totalmente pelo ambiente de desenvolvimento e o seu caso-de-uso pode ser representado pelo *workflow* da Figura 3.16.

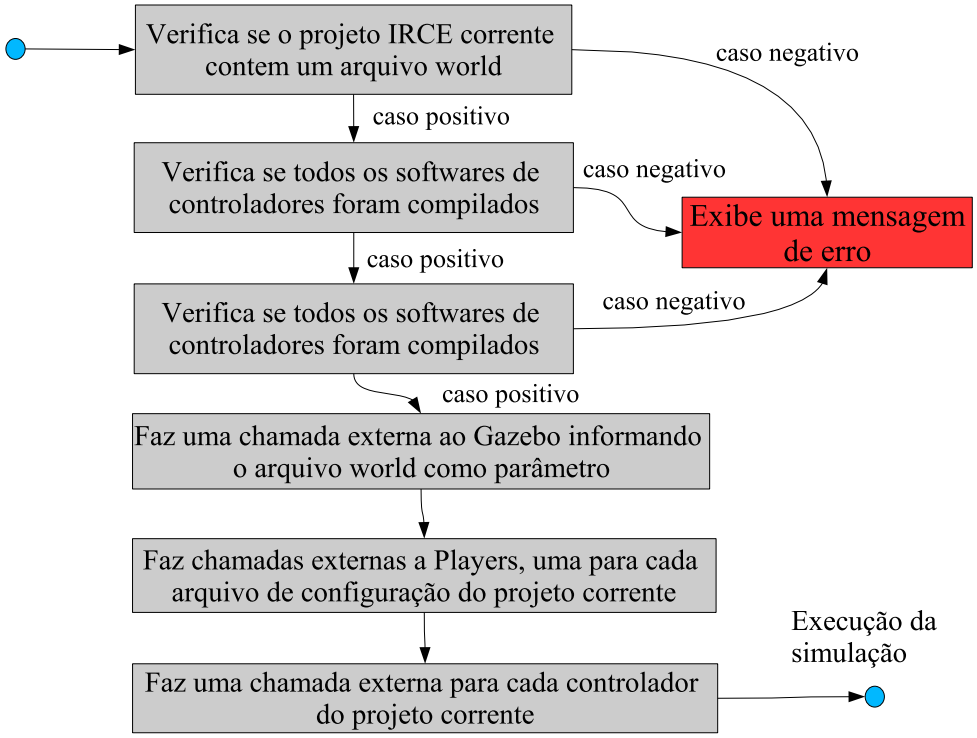


Figura 3.16: *Workflow* de simulação.

Através de apenas uma requisição, o sistema se encarrega de desempenhar todas as tarefas necessárias a simulação envolvendo o Gazebo com o mundo virtual, os servidor Player com suas configurações e os controladores.

3.3 Modelo de Classes de Análise

Ao mesmo tempo que os casos-de-uso são descritos, o modelos de classes de análise é capturado e evolui até atingir a uma estrutura de classes que consegue responder a todas as funcionalidades projetadas. O diagrama da Figura 3.17 mostra as classes capturadas nessa fase.

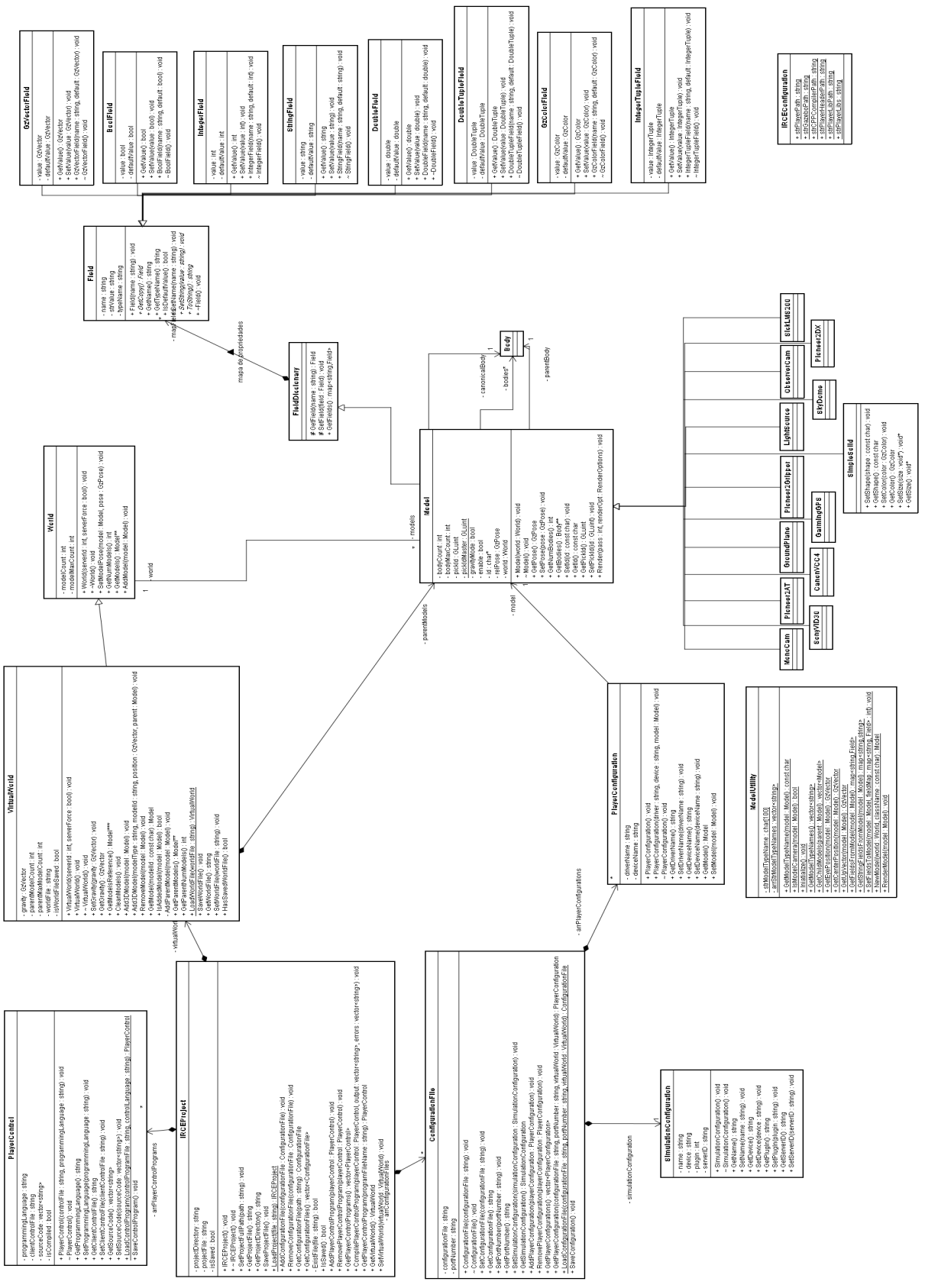


Figura 3.17: Modelo de classes de análise.

3.4 Projeto da Arquitetura do Sistema

Sendo o P/S/G um projeto de código-aberto que utiliza em sua grande maioria a linguagem C/C++ [26] e a biblioteca OpenGL [12] para a visualização do cenário no Gazebo, então logo no início do projeto do IRCE também são utilizadas as mesmas ferramentas computacionais para permitir o reuso de trechos de código ou estruturas de dados.

Por ser orientado a objetos, o sistema é composto por várias classes que assumem diferentes funções durante a execução do mesmo.

Uma das formas de se agrupar as classes no sistema é através da metodologia de desenho orientado a objeto de Coad & Yourdon [27] que consiste de quatro componentes:

- Componente de domínio do problema (CDP)
- Componente de interação humana (CIH)
- Componente de gerência de tarefa (CGT)
- Componente de gerência de dados (CGD)

De acordo com essa metodologia, as classes capturadas na fase de análise se encaixam na componente de domínio do problema e formam uma parte do modelo orientado a objeto de projeto. Por sua vez, as outras componentes formam o sistema computacional que vai suportar o *mini-mundo* provido pela componente de domínio do problema.

A componente de interação humana (CIH) é responsável pelas interações entre o usuário e o sistema e os também pelos detalhes dessas interações. A modelagem dessa componente contém o projeto das classes de interface gráfica.

A componente de gerência de tarefa (CGT) é responsável por definir e classificar as tarefas do sistema assim como fazer a comunicação e organização entre elas. A abordagem utilizada nesse projeto é modelar as classe de CGT como coordenadores de tarefas para a realização de um determinado de tarefas sobre um conjunto de objetos para dar forma a um caso-de-uso. Logo, para cada caso-de-uso capturado existe uma classe de tarefa correspondente.

A componente de gerência de dados (CGD) é a responsável pela infraestrutura de armazenagem e recuperação dos objetos de um sistema de gerência de dados que pode ser um banco-de-dados ou até mesmo arquivos.

Seguindo essa idéia, foram criados quatro pacotes [17], um para cada componente de Coad & Yourdon onde as classes estão agrupadas em pacotes de acordo com a responsabilidade que estas assumem dentro do sistema.

A Figura 3.18 mostra os pacotes do IRCE.

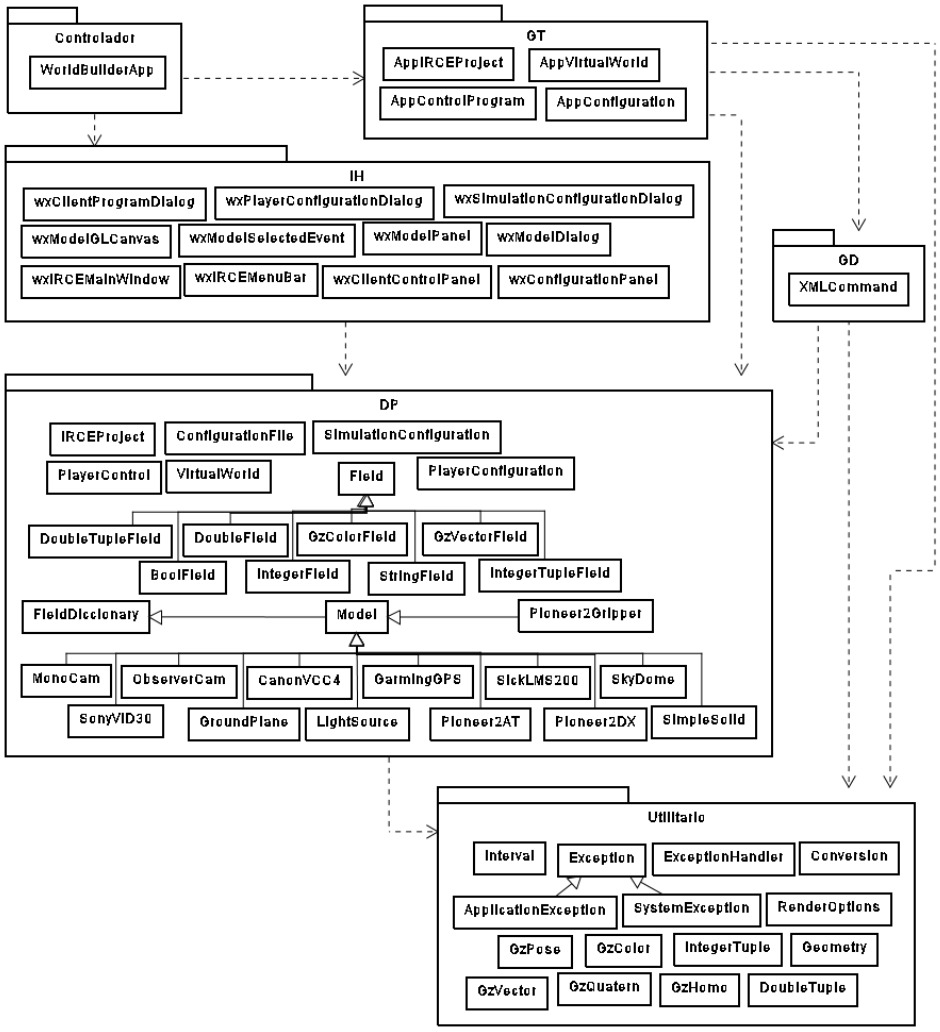


Figura 3.18: Diagrama de pacotes.

As classes de *interface* gráfica ficam no pacote IH, as classes de lógica da aplicação ficam no pacote GT e a única classe que faz o papel de controle fica no pacote Controlador.

A Figura 3.19 mostra a arquitetura do aplicativo que obedece a arquitetura MVC [28]. Ela separa as classes de *interface* gráfica que pertencem ao pacote IH das classes de lógica da aplicação que contêm a implementação dos requisitos no pacote GT. A intermediação é feita por uma classe controladora que faz a ponte entre esses dois grupos de classes que ficam no pacote Controlador.

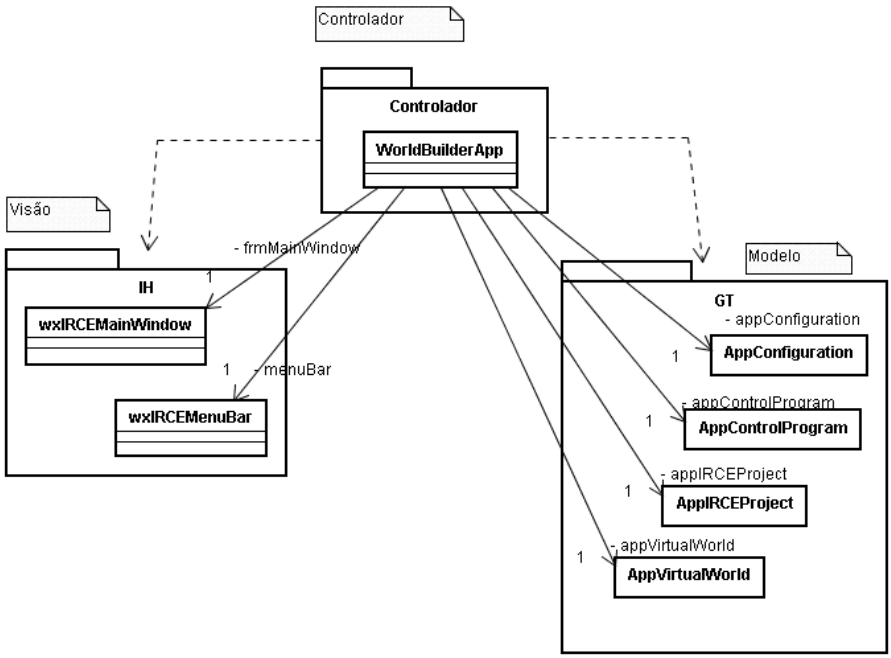


Figura 3.19: Arquitetura em MVC do IRCE.

3.4.1 Pacote DP

Conforme mostrado na Figura 3.17, este pacote contém as classes fundamentais do problema ao qual o sistema se propõe a resolver conforme capturado nos requisitos da seção 3.2.

3.4.2 Pacote GT

Esse pacote, mostrado na Figura 3.20, contém as classes de gerência de tarefa, ou seja, as classes que implementam os cenários capturados como casos-de-uso.

Para cada caso-de-uso existe uma classe nesse pacote que implementa a sua funcionalidade e cada método implementa um cenário do caso-de-uso.

3.4.3 Pacote IH

Esse pacote, mostrado na Figura 3.21, contém as classes que interagem com o usuário, ou seja, são as *interfaces* gráficas. Essas *interfaces* gráficas consistem num grupo de janelas e componentes gráficos que fazem uso da biblioteca *wxWidgets* [14].

- **wxIRCEMainWindow**: Janela principal da aplicação. É composta de um menu e de vários painéis. Cada painel é visualizado como uma aba.

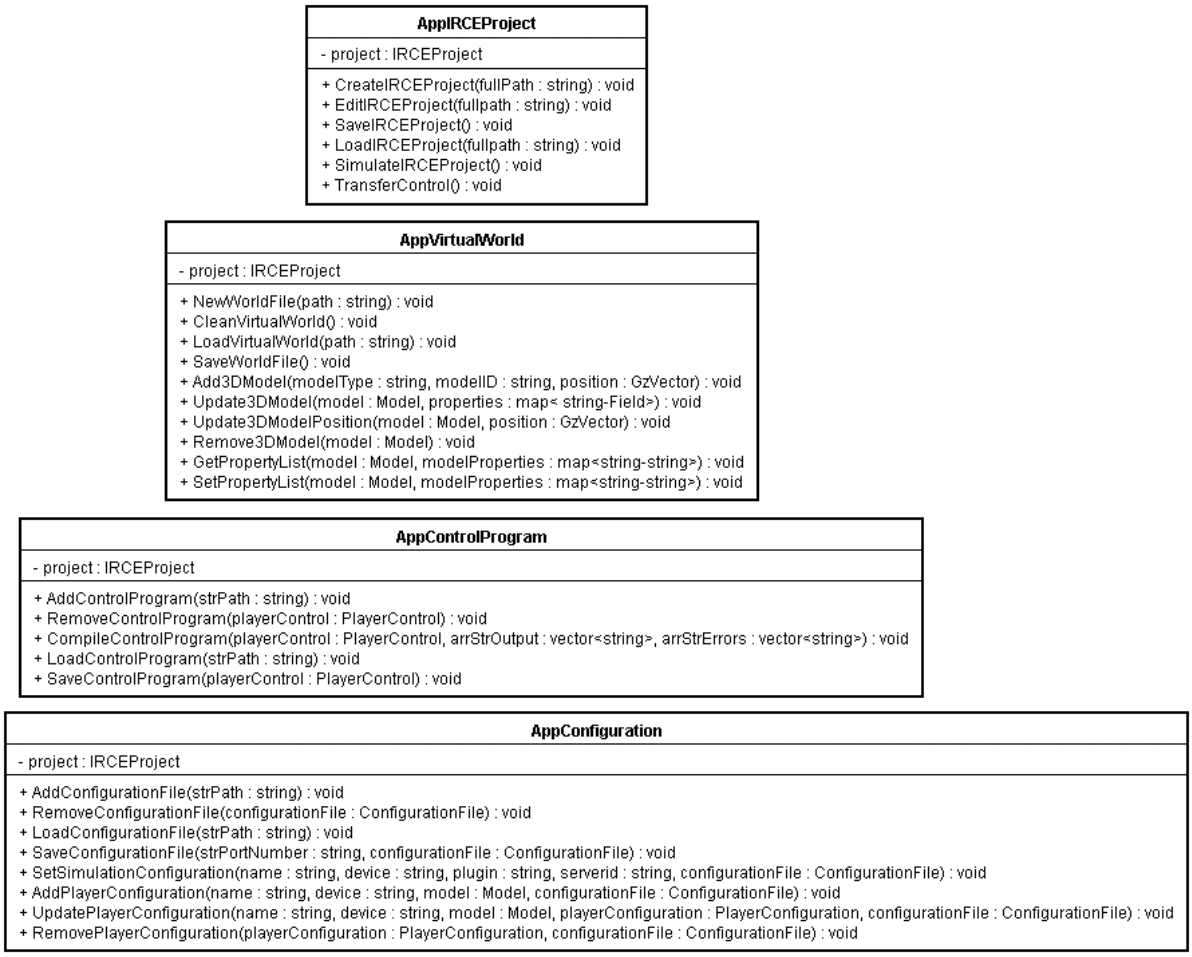


Figura 3.20: Diagrama de classes do pacote GT (Gerência de Tarefa).

- **wxIRCEMenuBar**: Menu principal da aplicação.
- **wxModelPanel**: Tipo de painel para a visualização do mundo virtual e a manipulação do mesmo.
- **wxClientControlPanel**: Tipo de painel para o código-fonte de um programa de controle.
- **wxConfigurationPanel**: Tipo de painel para visualizar um arquivo de configuração.
- **wxModelGLCanvas**: Componente gráfico para a visualização de uma lista de modelos 3D que utilizam o padrão OpenGL [12] para a sua renderização.
- **wxModelSelectedEvent**: Evento disparado pelo componente wxModelGLCanvas quando um modelo do mundo virtual é selecionado com o botão direito do mouse.
- **wxModelDialog**: Janela de diálogo para a adição de novos modelos no mundo virtual.

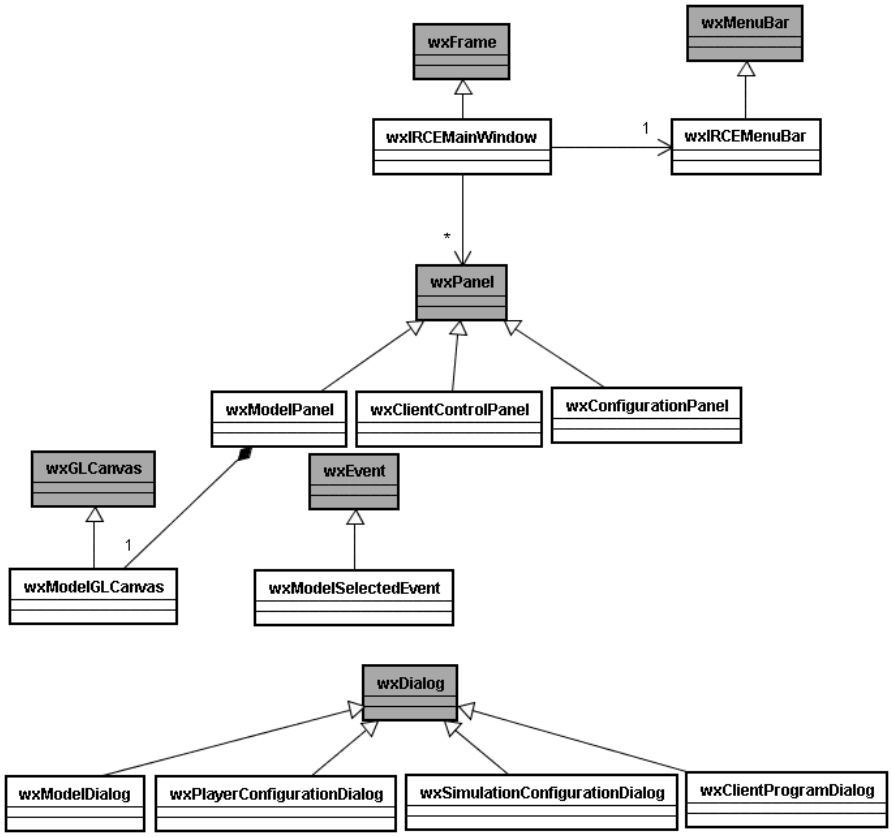


Figura 3.21: Diagrama de classes do pacote IH (*Interface* Humana).

- **wxPlayerConfigurationDialog**: Janela de diálogo para criar ou alterar uma configuração de um modelo.
- **wxSimulationConfigurationDialog**: Janela de diálogo para criar ou alterar uma configuração de simulação.
- **wxClientProgramDialog**: Janela de diálogo para selecionar um arquivo-fonte de programa de controle.

3.4.4 Pacote Controlador

Conforme mostrado na Figura 3.18, esse pacote contém uma classe que faz o papel de controladora, obedecendo ao padrão MVC (Model-View-Controller) [28]. Essa classe é responsável por capturar os dados informados na *interface* gráfica e passá-los para a camada de aplicação a fim de serem processados.

Por isso, ela tem uma referência para cada classe do pacote GT e uma referência para a janela principal da aplicação.

3.4.5 Estendendo o IRCE com Novos Modelos

A Figura 3.9 mostra alguns modelos do Gazebo que foram alterados para funcionar no sistema, entretanto, outros modelos podem ser adicionados para serem utilizados nesse ambiente. Para isso, o código do novo modelo deve ser alterado conforme os passos adiante:

1. Fazer a classe do modelo herdar de FieldDiccionary.
   Ex:

```
class NovoModelo : public FieldDiccionary
```

2. Alterar o construtor do modelo adicionando as propriedades do mesmo. Uma instancia de um tipo de propriedade específico é usada de acordo com o tipo do dado no modelo. Se alguma dessas propriedades afetar a forma como o modelo é desenhado, então uma alteração no método *Render* também é ser feita.
   Ex:

```
NovoModelo::NovoModelo(World * world)
{
  this->SetField(new GzVectorField("Propriedade1",valorDefault1));
  this->SetField(new StringField("Propriedade2","default"));
  ....
}
```

3.5 Mapa para Implementação

Os casos-de-uso descritos acima representam os objetivos que o sistema deve atingir e as classes fornecem a infra-estrutura de *software* necessária para que o sistema seja construído.

Dessa forma, os casos-de-uso e as classes fornecem uma especificação para a implementação do sistema, o que é mostrado no Capítulo 4.

Ao final, é mostrado como adicionar novos modelos já existentes no Gazebo mas que ainda não estão implementados no IRCE.

4 Projeto de Interface Homem-Máquina

O IRCE deve possuir seis funcionalidades principais:

- Gerência de Projeto IRCE;
- Montagem do Mundo Virtual;
- Configuração de Robô e seus Sensores;
- Programação de Controle;
- Simulação.

Para facilitar o uso do sistema pelo usuário, a interface gráfica, mostrada na Figura 4.1, segue a analogia dos ambientes de desenvolvimento *RAD* (*Rapid Application Development*) assim como o Eclipse [22].

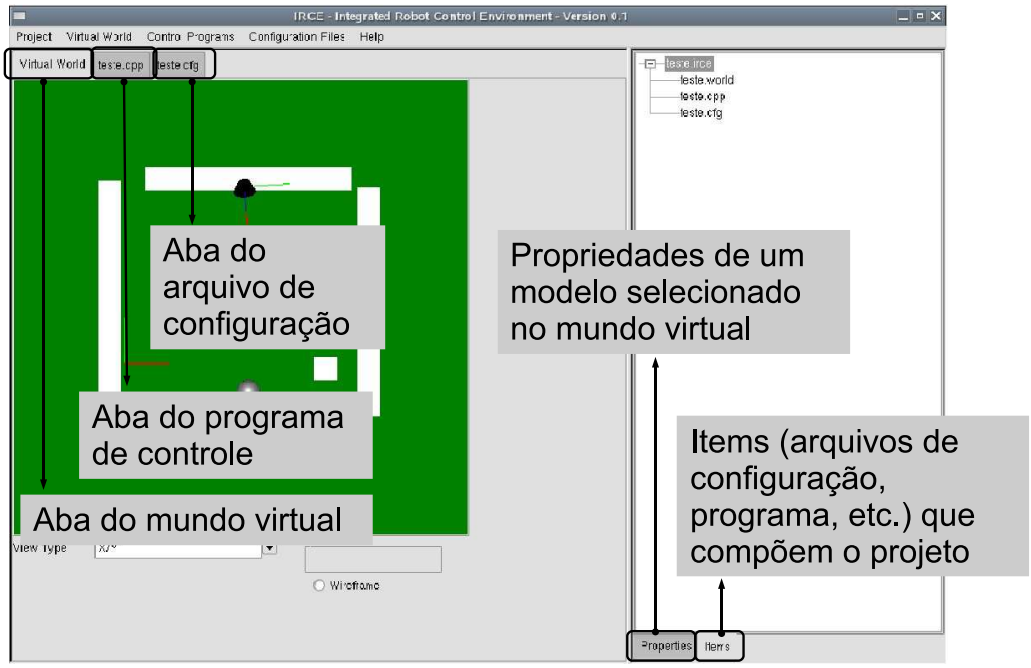


Figura 4.1: Janela principal do IRCE.

Um projeto de simulação de robôs é composto por um mundo virtual, vários arquivos de configuração e vários arquivos de programa. Cada um desses elementos pode ser visualizado através de abas na janela principal.

4.1 Gerência de Projeto IRCE

Para melhor gerenciar um projeto de simulação de robôs é criado o arquivo de projeto (*.irce).

Esse arquivo de projeto é o ponto inicial na utilização do sistema, pois é a partir dele que os outros arquivos poderão ser trabalhados. Ele contém uma referência para um arquivo (*.world), várias referências para arquivos de configuração e várias referências para arquivos fonte de controladores.

O arquivo (*.irce) segue o padrão XML e relaciona os arquivos que fazem parte do projeto. Os arquivos que fazem parte do projeto são definidos por meio de *tags* conforme mostrado no exemplo abaixo:

```
<irce>
    <worldFile>mundo_virtual.world</worldFile>
    <configurations>
        <configurationFile>robo1.cfg</configurationFile>
        <configurationFile>robo2.cfg</configurationFile>
    </configurations>
    <controls>
        <controlFile>controlador1.cpp</controlFile>
        <controlFile>controlador2.cpp</controlFile>
    </controls>
</irce>
```

Todas as operações de gerência de projeto (criar, carregar, salvar e editar) estão no menu principal no item Project, na parte direita. Ao executar o IRCE, esse é o primeiro menu a ser habilitado pelo usuário, pois é a partir do projeto que os outros itens podem ser criados.

Por exemplo, usando esse menu, para criar um projeto IRCE, o caminho (**Project -> Create Project**) deve ser executado e um diálogo, como na Figura 4.2, aparece para requisitar o caminho no diretório de arquivos do sistema onde o arquivo de projeto deve ser gravado.

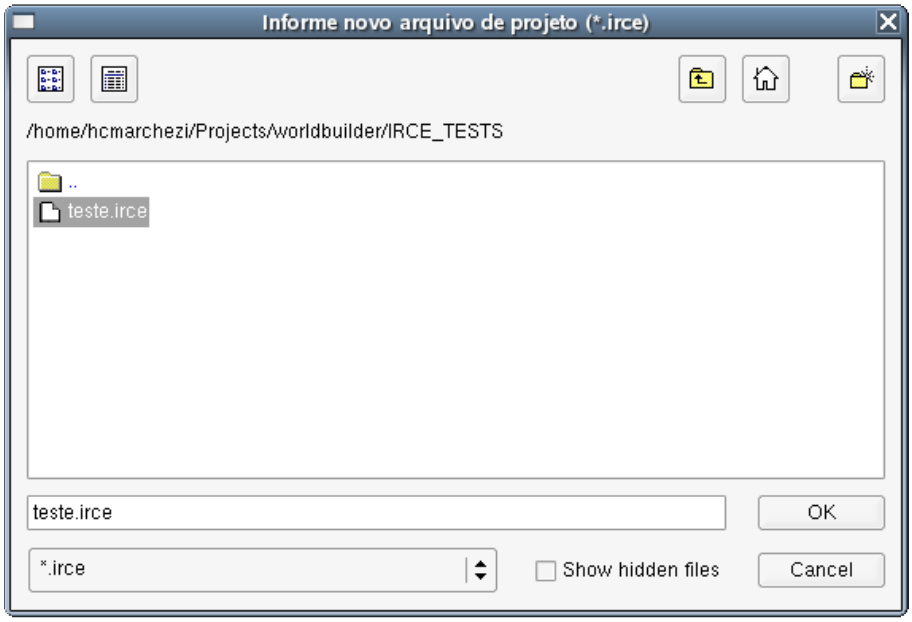


Figura 4.2: Janela de diálogo para criação de projeto.

4.2 Montagem do Mundo Virtual

Para a montagem do mundo virtual, é preciso, além de criar o projeto, de um arquivo de mundo virtual para o projeto que pode ser criado ou pode ser carregado de um arquivo pré-existente.

As classes em C++ que representem modelos 3D no Gazebo são reutilizadas e fazem parte da implementação do IRCE. Cada modelo possui propriedades que descrevem características físicas e/ou comportamentais que podem ser configuradas na *tag* que declara o modelo via arquivo *world*.

Essas propriedades são representadas como atributos comuns encapsulados pelas classes, porém isso torna muito difícil, se não impossível, de se obter informação organizada sobre essas propriedades para um determinado modelo genérico em tempo de execução, ou seja, ao selecionar um modelo do mundo virtual, o sistema não sabe quais as propriedades desse modelo e logo não há como listá-las na *interface* gráfica para serem visualizadas e alteradas. Devido aos modelos do Gazebo serem classes em C++ ,estes não possuem meta-dados ao serem compilados para o sistema como em outras linguagens tais como Java ou Python.

Para resolver esse problema, todos os atributos de um modelo são representados numa estrutura de mapa [29] disponível na Standard Template Library [30], que faz parte da linguagem C++, conforme mostrado na Figura 3.10.

Seguindo essa idéia, a classe *Model*, disponibilizada pelo Gazebo, passa a herdar da classe

FieldDiccionary que disponibiliza um mapa de propriedades. O segundo passo é alterar cada construtor dos modelos específicos adicionando as propriedades que fazem parte do determinado modelo. Como todos os modelos utilizam essa estrutura de dados, ao selecionar um, é possível obter uma lista de propriedades (*class Field*) contendo o nome e o valor de cada propriedade, e pode ser usada para visualizar as propriedades dinamicamente. A outra vantagem de utilizar um mapa de propriedades para modelos é a serialização e deserialização para um arquivo XML sem que seja necessária qualquer alteração nesses modelos, que não devem saber como isso é feito, mas sim a classe *XMLCommand*.

A classe *XMLCommand*, veja Figura 3.11, serializa qualquer objeto que herde da classe Model fazendo uso das propriedades registradas. Cada *tag* é nomeada através do nome da propriedade e o valor da propriedade é convertido em string. A saída é um objeto *TiXmlElement*, que faz parte da biblioteca TinyXML [16] e representa um elemento de um arquivo XML. Na deserialização, o processo inverso ocorre, um elemento de arquivo XML é informado e este é processado e convertido num objeto Model genérico. A montagem consiste em adicionar, remover ou alterar os modelos 3D que compõem um mundo virtual. Após a adição do mundo virtual ao projeto, no mesmo menu Virtual World agora é possível ver os sub-itens para manipulação de modelos 3D habilitados.

Para criarmos um mundo virtual, deve ser selecionado o menu (**Virtual World -> New World File**). O sistema mostra uma janela de diálogo para informar o caminho onde o arquivo World é gravado como mostrado na Figura 4.3.

Figura 4.3: Janela de diálogo para criação de arquivo world.

A partir do mundo virtual criado pode-se adicionar modelos 3D selecionando (**Virtual World -> Add 3D Model**). Um diálogo aparece para informar dados básicos sobre um modelo 3D tais como:

*** Tipo:** Se é um Pioneer2DX, Pioneer2AT, CanonVCC4, SimpleSolid (um obstáculo estático), etc.

*** Identificação:** um nome unicamente identificado dentro do mundo virtual.

*** Posição:** uma posição no espaço 3D onde o modelo vai aparecer.

Este diálogo é mostrado na Figura 4.4.

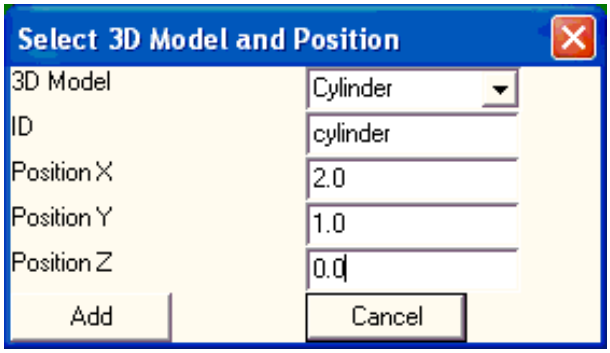


Figura 4.4: Janela de diálogo para adicionar um modelo 3D no mundo virtual.

A inserção de modelos é feita somente através da janela de diálogo onde o desenvolvedor informa a posição inicial do modelo mas para aumentar a interatividade na construção do mundo virtual, após inseridos, os modelos podem ser selecionados e movidos no ambiente virtual com o uso do *mouse*.

A Figura 4.5 mostra que no IRCE também é possível visualizar e alterar dados das propriedades que compõem um modelo selecionando-o no painel de visualização, assim como numa ferramenta RAD. Para que as alterações tenham efeito, é necessário pressionar o botão **Update** no painel de propriedades.

4.3 Configuração de Robô e seus Sensores

Para que o Player consiga controlar os vários tipos de *hardware*, é necessário editar um arquivo de configuração. O arquivo de configuração é responsável por instanciar os *drivers* necessários para controlar o robô e também especificar como acessar o *hardware*. Em outras palavras, o arquivo de configuração mapeia os dispositivos físicos (*hardware*) em dispositivos do Player (*interface*).

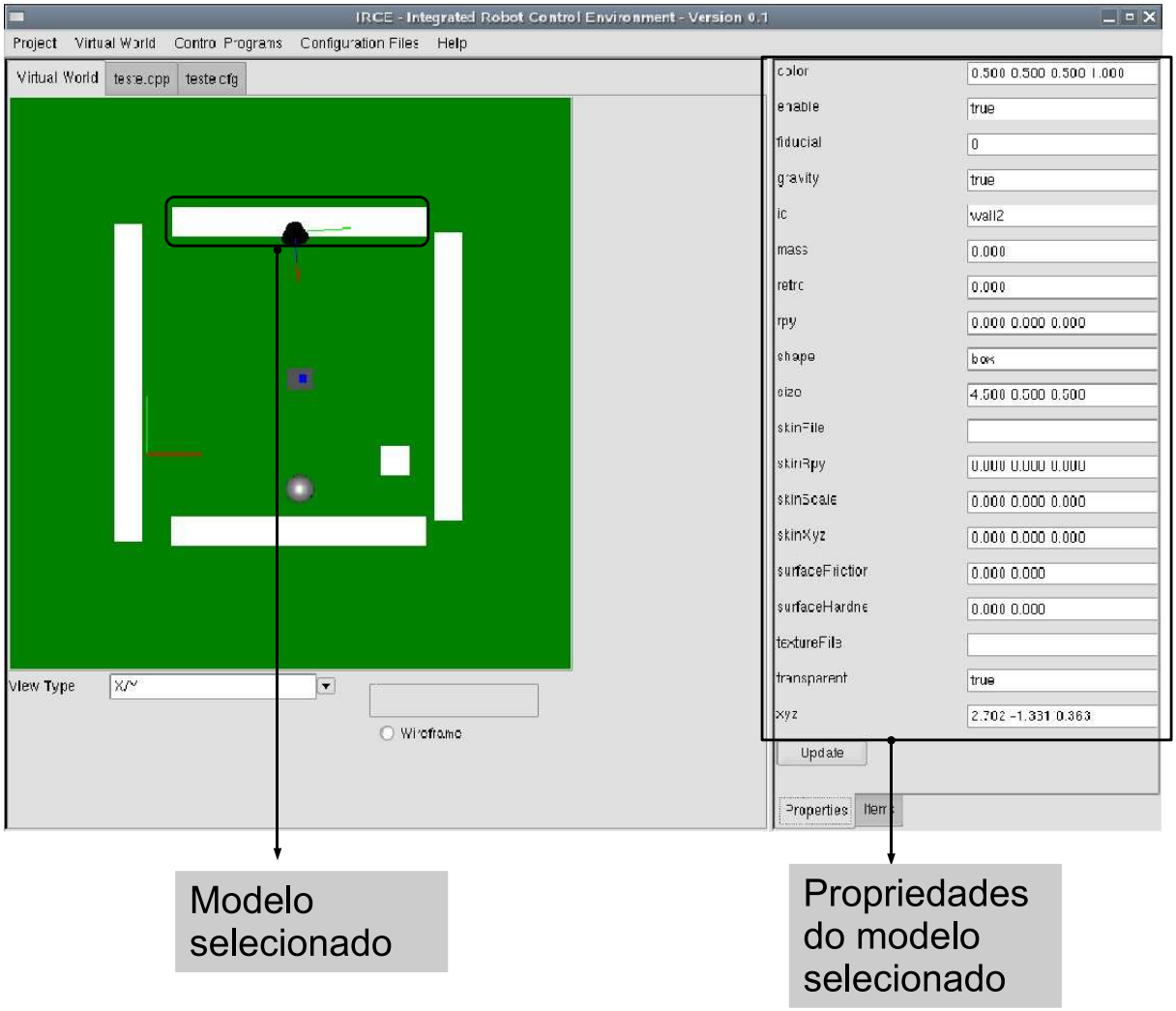


Figura 4.5: Painel de mundo virtual em 3D e as propriedades de um modelo 3D selecionado.

Conforme explicado na Seção 2.2.2, o arquivo de configuração é composto por seções com o cabeçalho *driver* no qual o dispositivo é declarado e ligado ao seu respectivo *driver*. Ao utilizar o Gazebo para simulação, o arquivo de configuração deve conter uma seção *driver* para configurar o plug-in e outras seções *driver* para configurar o robô e cada dispositivo (sensor, garra, etc.) utilizado por ele. Por isso, o arquivo de configuração é representado por uma classe que contém uma configuração de servidor e várias configurações de dispositivos, o que pode ser visto no diagrama de classes da Figura 3.13.

Esses arquivos podem ser trabalhados pelo menu **Configuration**. A Figura 4.6 mostra uma aba para configuração de dispositivos, como sensores, por exemplo, normalmente ligados a um robô.

Ao selecionar o menu (**Configuration -> Add Configuration File**), o diálogo da Figura 4.7 aparece onde o sistema requisita o caminho do arquivo de configuração no sistema.

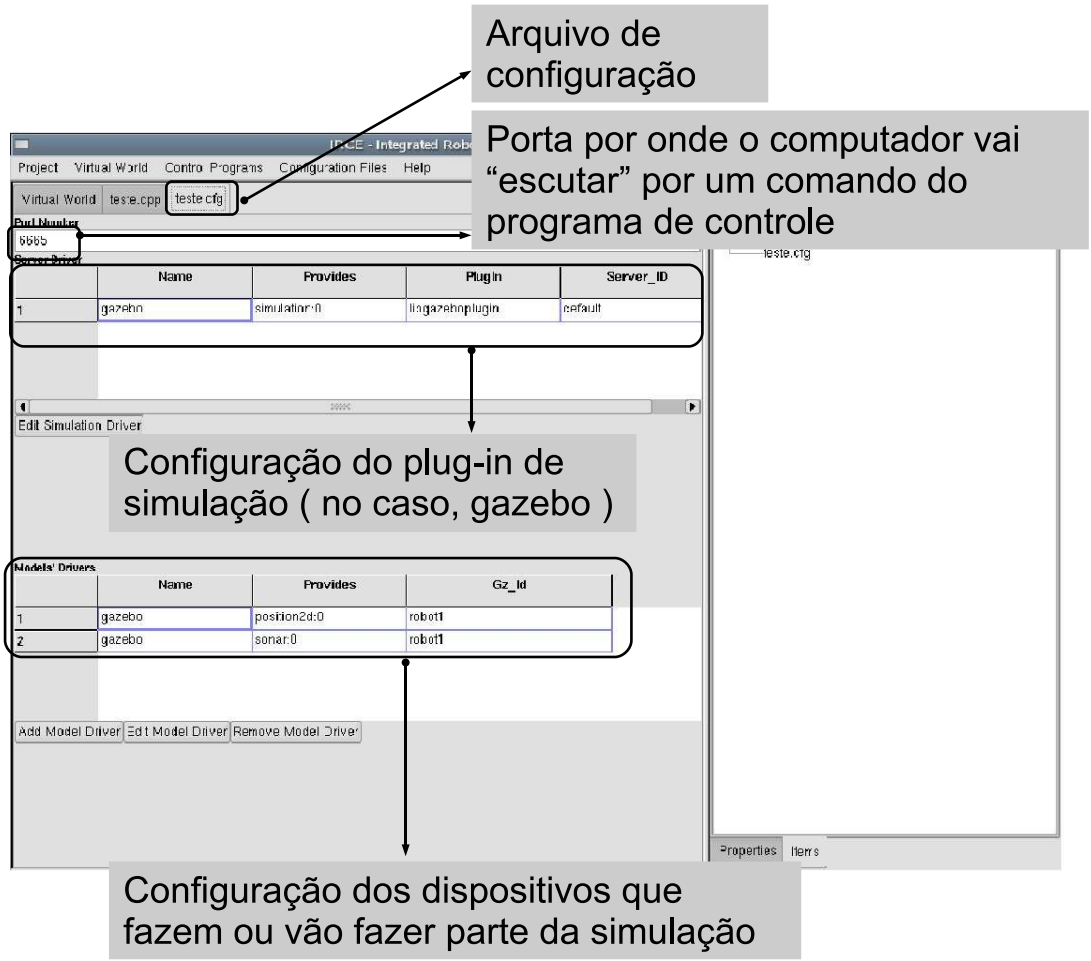


Figura 4.6: Painel de configuração de dispositivos.

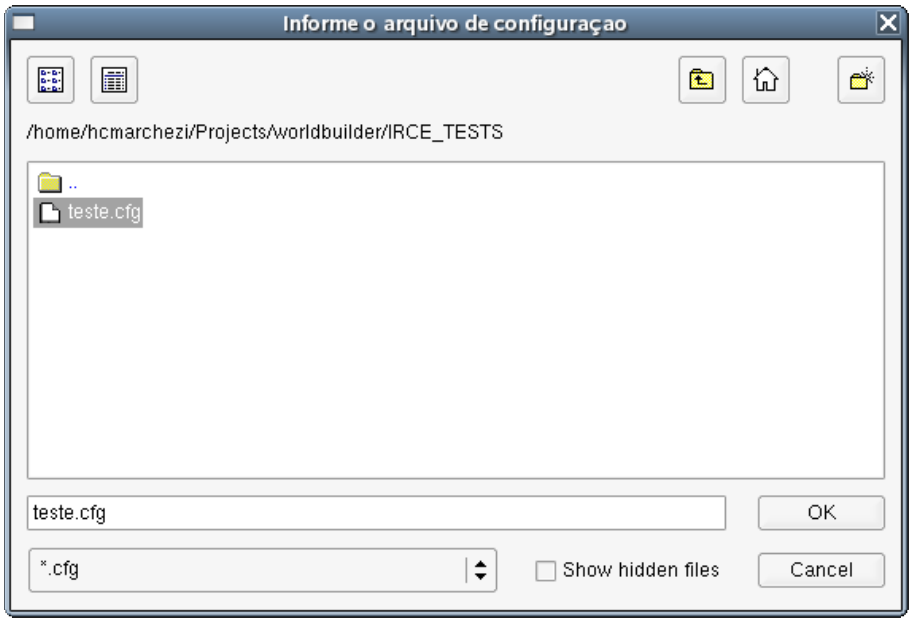


Figura 4.7: Janela de diálogo para a criação de arquivo de configuração.

4.4 Programação de Controle

Por fim, após adicionar um modelo 3D num mundo virtual e configurá-lo devidamente no arquivo de configuração, é necessário desenvolver um programa de controle para esse modelo.

O código do controlador deve ser compilado através de bibliotecas de desenvolvimento de *software* para controlador. Embora tais bibliotecas estejam disponíveis em C, C++, Java, entre outras, por motivos de simplicidade, a linguagem C++ é a única utilizada por esse sistema.

A compilação é feita através de uma chamada externa para o compilador g++, que está normalmente disponível no sistema operacional LINUX. Para saber se a compilação ocorreu com erros ou não, as mensagens de saída do compilador são capturadas e mostradas numa área separada na *interface* gráfica.

Arquivos de programas podem ser adicionados, removidos ou compilados no menu (**Control Programs**).

Ao executar (**Control Programs -> Add Program File**), a janela de diálogo da figura 4.8 abre requisitando informar o caminho no sistema de arquivo, onde o arquivo do programa de controle em C++ com extensão cpp deve ser gravado.

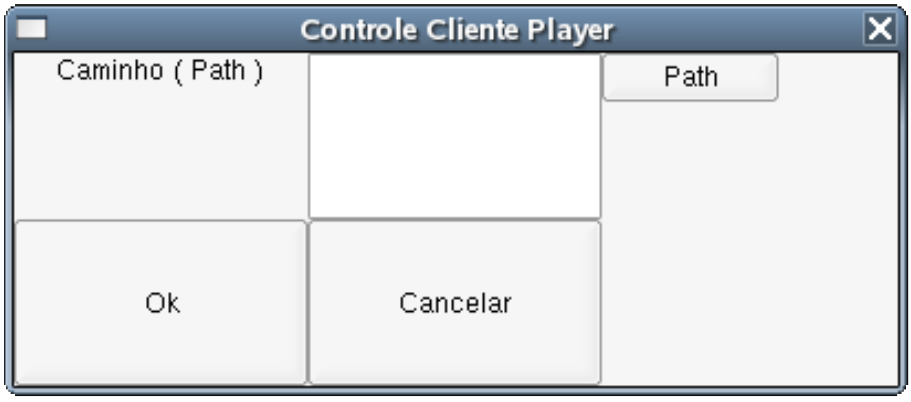


Figura 4.8: Janela de diálogo para a criação de programa de controle.

Também é possível compilar o programa de controle no mesmo ambiente de desenvolvimento, o que pode ser feito através do menu (**Control Programs -> Compile Program File**). A Figura 4.9 mostra a aba para programação de controle para um robô. O resultado da compilação, incluindo eventuais erros durante a implementação do código, pode ser visto no painel da parte de baixo de onde o programa de controle é editado.

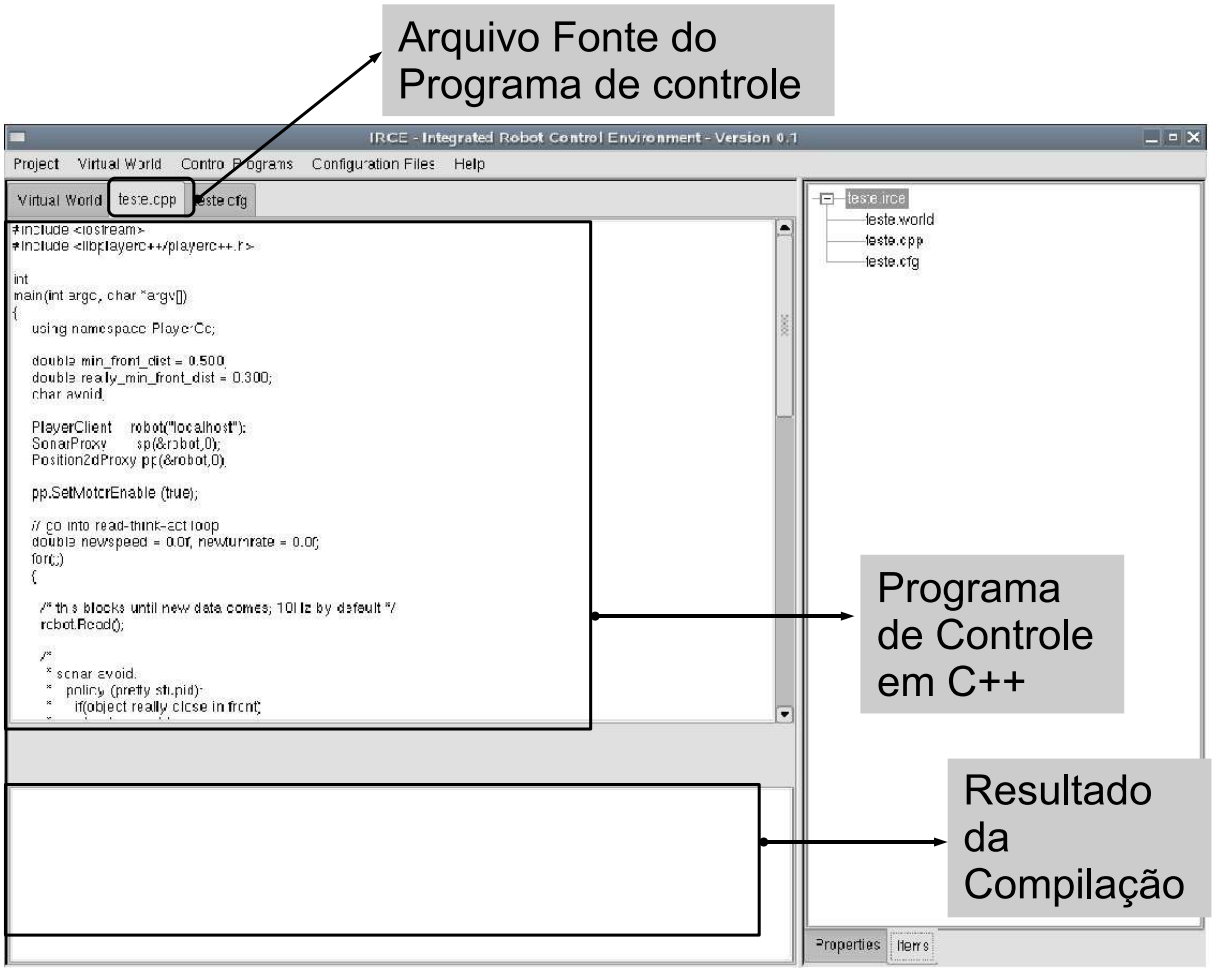


Figura 4.9: Painel para programação de controle dos robôs.

4.5 Simulação

Finalmente o próximo passo é executar uma simulação para verificar o resultado do programa de controle.

Os aplicativos Gazebo, Player e os controladores são invocados através de processos independentes. A partir daí, o IRCE passa o controle de eventos para esses aplicativos. A simulação pode ser interrompida através de uma janela de diálogo com o botão para interromper, daí o controle de eventos volta para o IRCE.

Para simular um projeto IRCE, só é necessário executar o menu (**Project -> Simulation**). A Simulação é executada por uma chamada externa ao aplicativo Gazebo, passando o mundo virtual do projeto e tantos Player quantos forem os arquivos de configuração. Para interromper a simulação, basta clicar no botão **Ok** numa janela de diálogo que fica ativa durante toda a simulação. A Figura 4.10 mostra o IRCE executando uma simulação simples.

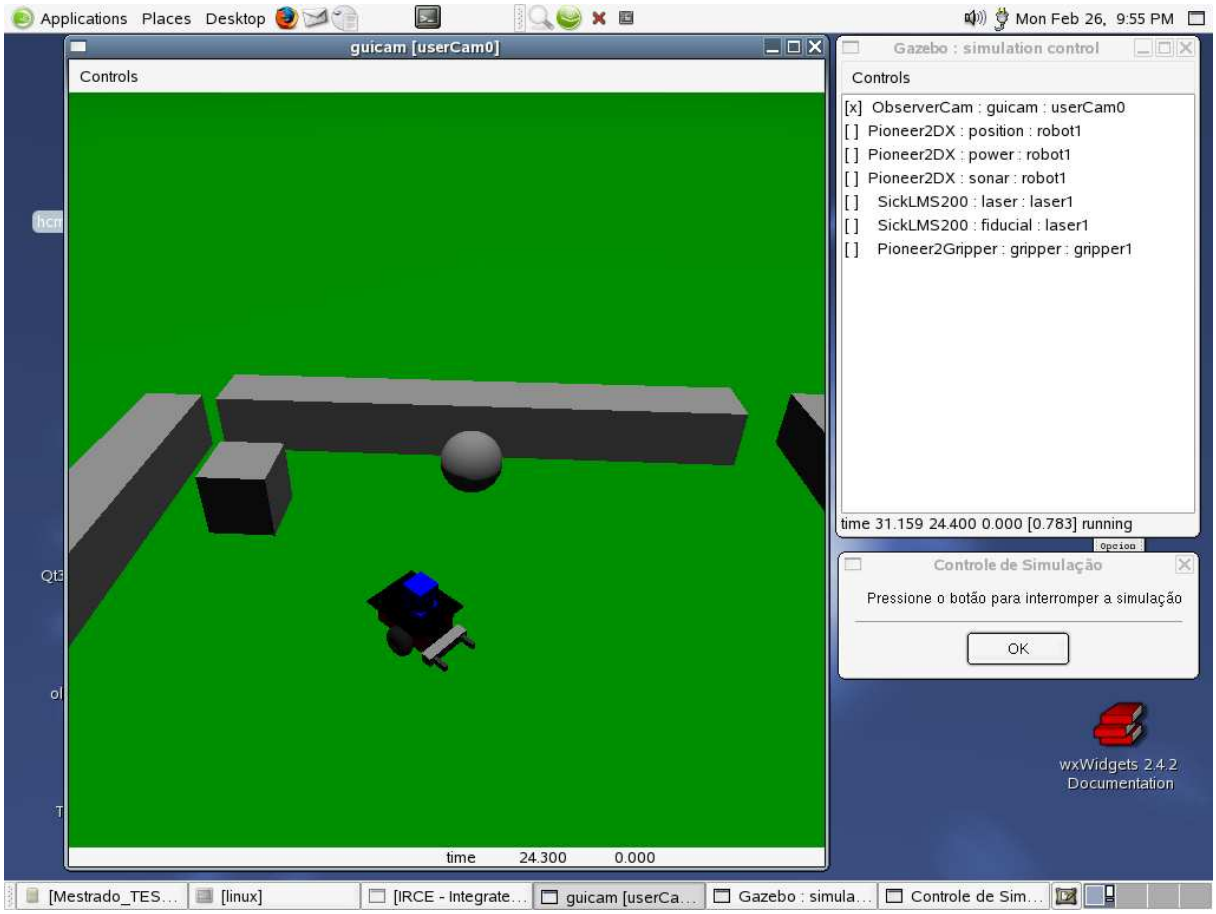


Figura 4.10: Simulação envolvendo um robô é executada no IRCE.

4.6 Protótipo

Este capítulo mostra como os casos-de-uso e as classes capturados na fase de desenvolvimento foram efetivamente implementados no sistema.

As funcionalidades das seções anteriores são demonstradas através de um protótipo do ambiente de desenvolvimento.

O protótipo é funcional pois preenche todos os requisitos listados acima. Algumas melhoras podem ser feitas na parte de edição do código do programa controlador, incluindo realce de palavras reservadas da linguagem de programação utilizada, assim como nos ambientes integrados de desenvolvimento atuais.

De qualquer forma, vários modelos tridimensionais do Gazebo já foram implementados no sistema, o que permite que o usuário consiga realizar várias simulações.

5 *Exemplos de Uso do IRCE*

O exemplo a seguir mostra como usar o IRCE para melhor ilustrar a utilização do ambiente. Para isso, um passo-a-passo ilustra cada etapa até à fase de simulação.

1. Execute o IRCE.
2. Crie um projeto IRCE (**Project -> New Project File**) informando um caminho e um nome para o arquivo de projeto (ex: exemplo.irce)(Figura 5.1).

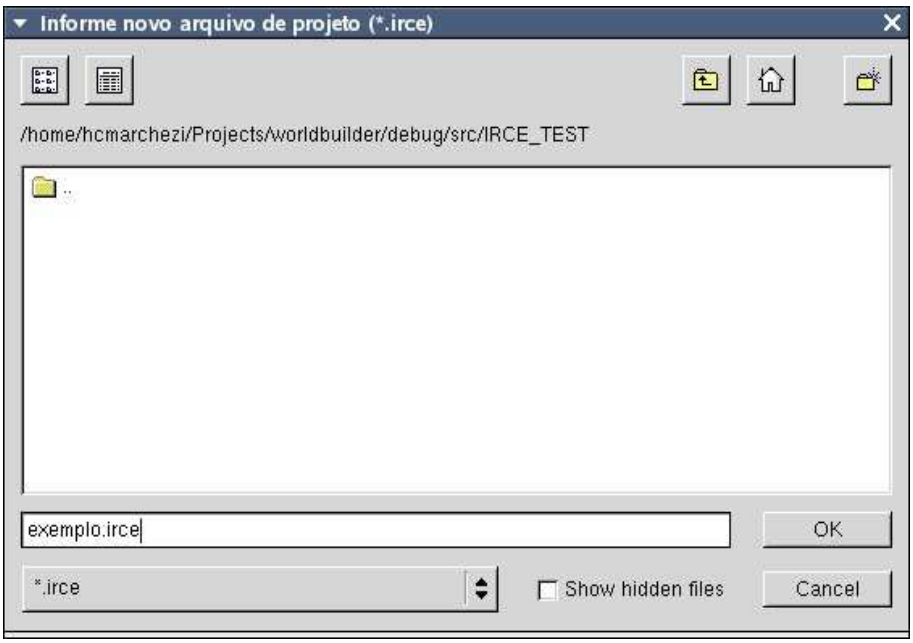


Figura 5.1: Projeto exemplo.irce é criado.

3. Crie um mundo virtual (**Virtual World -> New World File**) informando um caminho e um nome para o mundo virtual (ex: exemplo.world)(Figura 5.2).

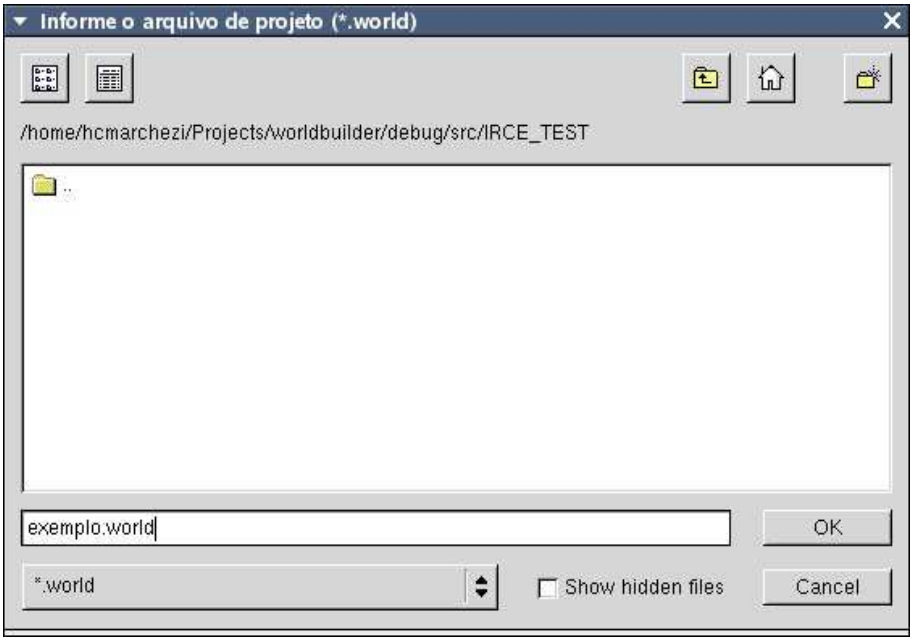


Figura 5.2: Mundo virtual exemplo.world é criado.

4. Adicione um modelo 3D (**Virtual World -> Add 3D Model**), informando os dados, conforme a Figura 5.3.

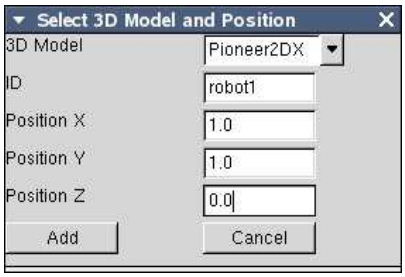


Figura 5.3: Modelo Pioneer 2DX é adicionado no mundo virtual.

5. Repita o mesmo procedimento para outros modelos 3D (**Virtual World -> Add 3D Model**), informando os dados de acordo com a tabela 5.1. O mundo virtual ficará conforme mostrado na Figura 5.4.

Type	**Id**	**Position**
GroundPlane (chão)	piso	(0.0, 0.0, 0.0)
Box (caixa)	parede1	(-3.0,0.0,0.0)
Box	parede2	(3.0, 0.0, 0.0)
Box	parede3	(0.0,-3.0, 0.0)
Box	parede4	(0.0,3.0, 0.0)
Pioneer2AT	robot2	(-1.0, -1.0, 0.0)
LightSource	luz	(0.0,0.0, -10.0)
Type	**Id**	**RenderMethod**
ObserverCam (observador)	observador	GLX

Tabela 5.1: Modelos a serem adicionados ao mundo virtual e a sua posição inicial

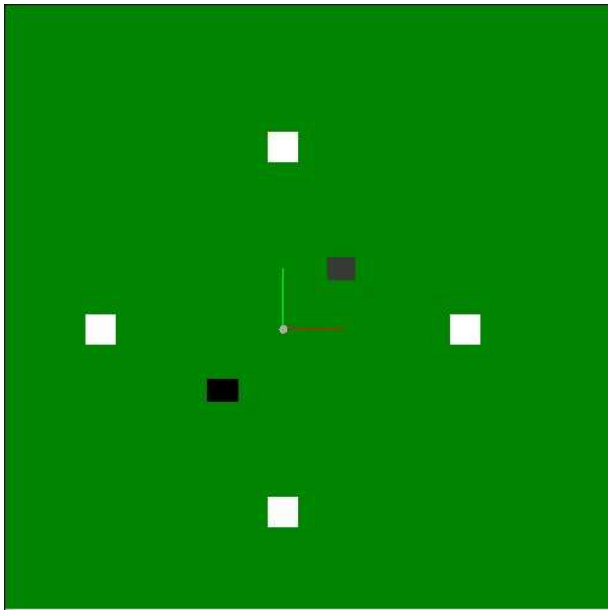


Figura 5.4: Mundo virtual após adicionar os modelos.

6. Clique na caixa branca à esquerda com o botão direito do mouse, altere a propriedade *size* para (0.5, 5.0, 0.5) e altere a propriedade *mass* para 20.0 . Pressione o botão **Update** e a parede vai se esticar na vertical. O resultado pode ser visto na Figura 5.5.

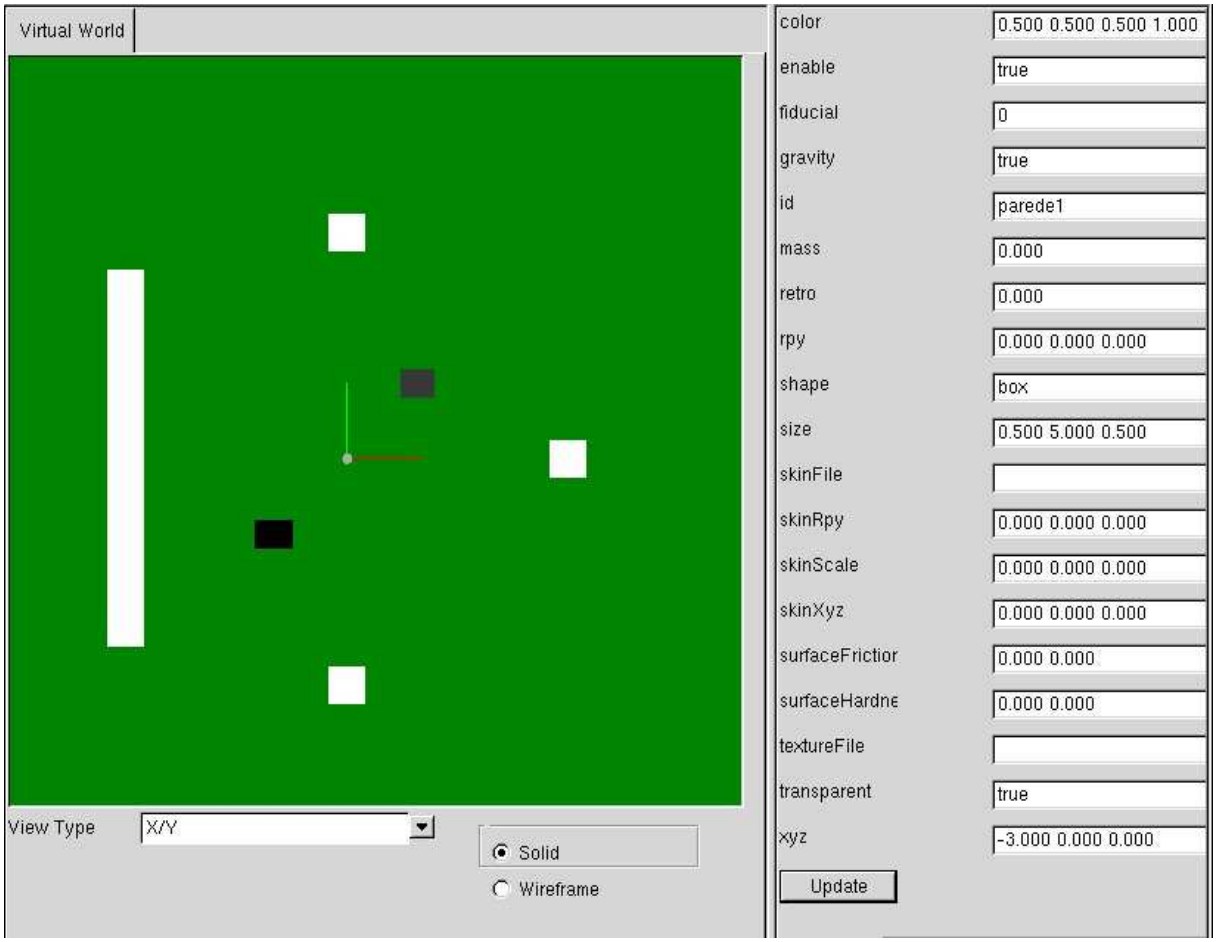


Figura 5.5: Mundo virtual com parede esticada e painel de propriedades ao lado.

7. Repita o mesmo procedimento para cada uma das outras caixas brancas, conforme mostrado na tabela 5.2. Ao final, o mundo virtual se parecerá, conforme na Figura 5.6.

Type	**Id**	**Position**
parede2	(0.5, 5.0, 0.5)	20.0
parede3	(5.0, 0.5, 0.5)	20.0
parede4	(5.0, 0.5, 0.5)	20.0

Tabela 5.2: Propriedades a serem alteradas nas outras paredes(caixas brancas).

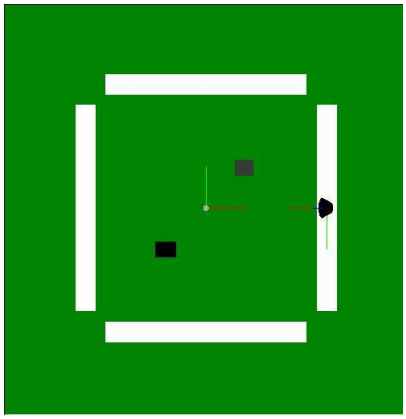


Figura 5.6: Mundo virtual após informar propriedades.

8. Adicione um arquivo de configuração no mesmo caminho do arquivo de projeto, mas com o nome *robot1.cfg*. (**Configuration Files -> Add Configuration File**)(Veja Figura 5.7)

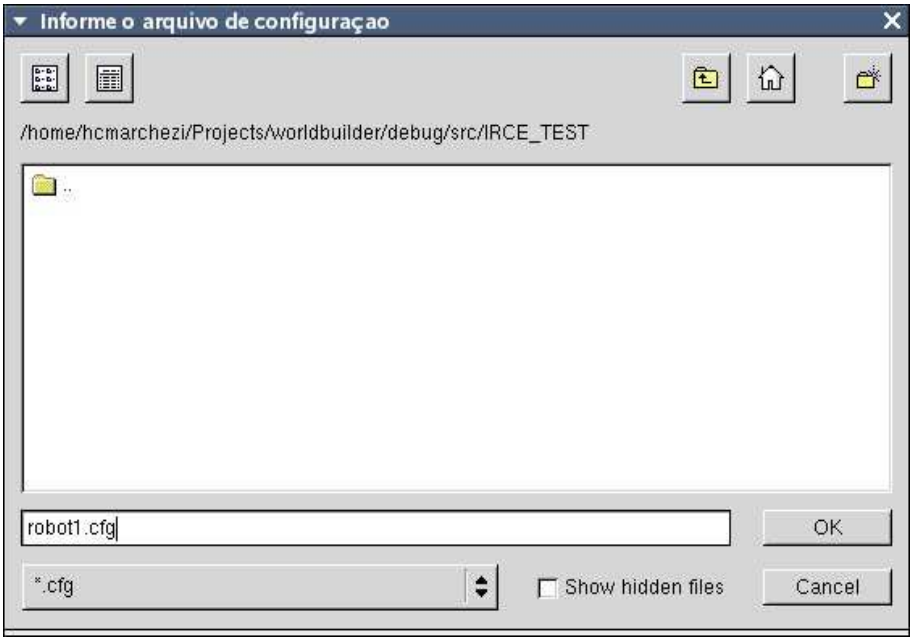


Figura 5.7: O arquivo de configuração robot1.cfg é adicionado no projeto.

9. Adicione outro arquivo de configuração no mesmo caminho com o nome robot2.cfg. (**Configuration Files -> Add Configuration File**)

10. Selecione a aba *robot1.cfg* e edite a configuração, conforme mostrado na Figura 5.8.

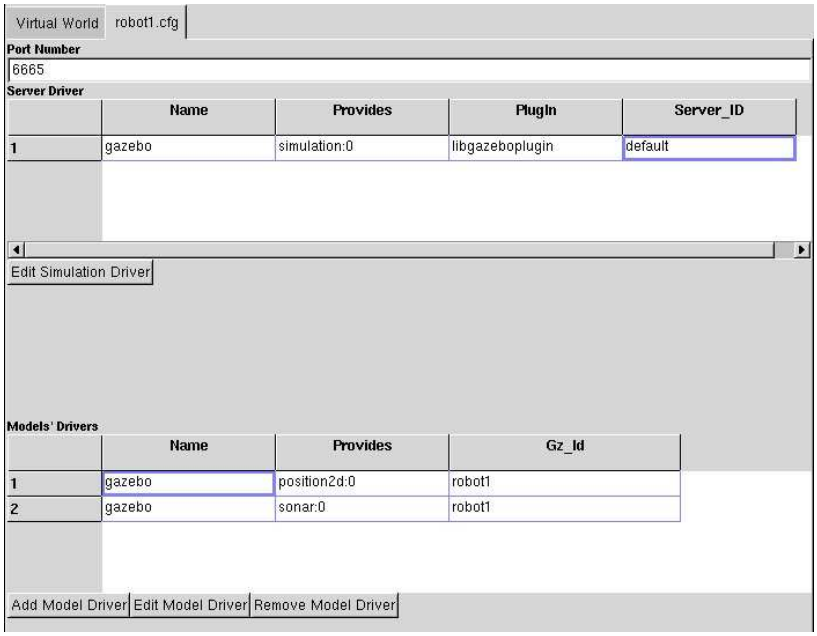


Figura 5.8: Dados do arquivo de configuração robot1.cfg .

11. Selecione a aba *robot2.cfg* e edite a configuração, conforme mostrado na Figura 5.9.

12. Adicione um programa de controle no mesmo caminho do arquivo de projeto e com o nome *robot1.cpp*, conforme mostrado na figura. (**Control Programs -> Add Control Program**)(Veja Figura 5.10)

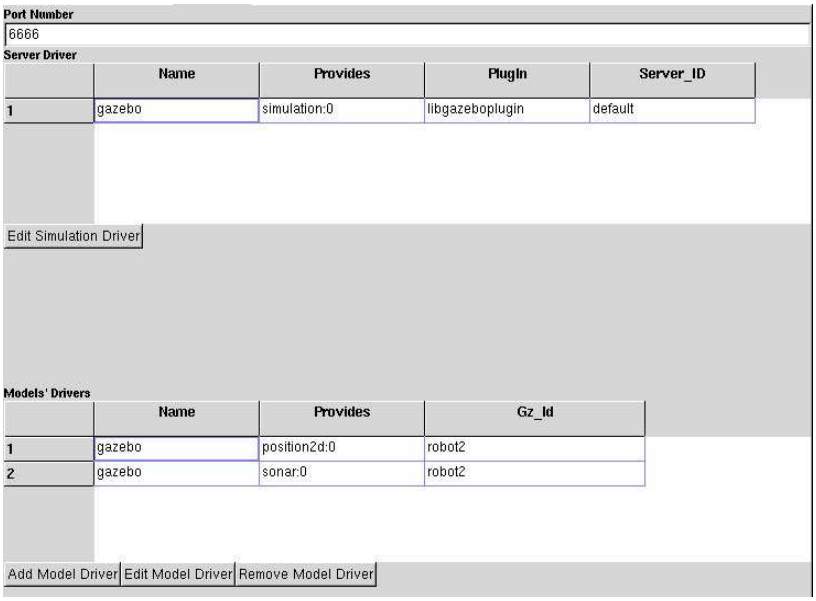


Figura 5.9: Dados do arquivo de configuração robot2.cfg .

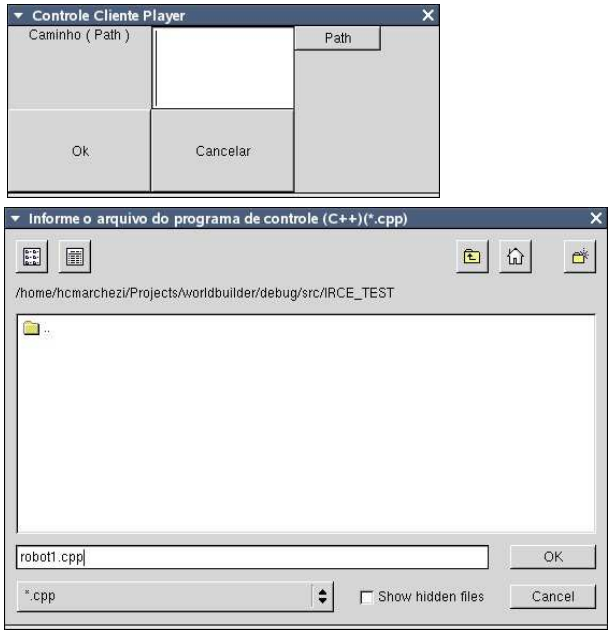


Figura 5.10: O programa do controlador de robot1 é adicionado ao projeto.

13. Adicione outro programa de controle no mesmo caminho mas com o nome *robot2.cpp*.
14. Selecione a aba *robot1.cpp* e digite o seguinte programa de controle:

```
#include <iostream>
#include <libplayerc++/playerc++.h>

int
main(int argc, char *argv[])
{
  using namespace PlayerCc;
```

```
  PlayerClient    robot("localhost",6665);
  SonarProxy      sp(&robot,0);
  Position2dProxy pp(&robot,0);

  for(;;)
  {
    double turnrate, speed;

    // read from the proxies
    robot.Read();

    // print out sonars for fun
    std::cout << sp << std::endl;

    // do simple collision avoidance
    if((sp[0] + sp[1]) < (sp[6] + sp[7]))
      turnrate = dtor(-20); // turn 20 degrees per second
    else
      turnrate = dtor(20);

    if(sp[3] < 0.500)
      speed = 0;
    else
      speed = 0.100;

    // command the motors
    pp.SetSpeed(speed, turnrate);
  }
}
```

15. Selecione a aba *robot2.cpp* e digite o seguinte programa de controle:

```
/*
 * randomwalk.cc - sonar obstacle avoidance with random walk
 */

#include <libplayerc++/playerc++.h>
#include <iostream>
using namespace PlayerCc;

double minfrontdistance = 0.750;
double speed = 0.200;
double avoidspeed = 0; // -150;
double turnrate = DTOR(40);

int main(int argc, char** argv)
{
  int randint;
  int randcount = 0;
  int avoidcount = 0;
  bool obs = false;
//  parse_args(argc,argv);
  SonarProxy *sp = NULL;
```

```
// we throw exceptions on creation if we fail
try
{

  PlayerClient robot("localhost",6666);
  Position2dProxy pp(&robot, 0);

  sp = new SonarProxy (&robot, 0);

  std::cout << robot << std::endl;

  pp.SetMotorEnable (true);

  // go into read-think-act loop
  double newturnrate=0.0f, newspeed=0.0f;
  for(;;)
  {

    robot.Read();


    obs = ((sp->GetScan (2) < minfrontdistance) ||   // 0?
           (sp->GetScan (3) < minfrontdistance) ||
           (sp->GetScan (4) < minfrontdistance) ||   // 0?
           (sp->GetScan (5) < minfrontdistance) );

    if(obs || avoidcount || pp.GetStall ())
    {
      newspeed = avoidspeed;

      /* once we start avoiding, continue avoiding for 2 seconds */
      /* (we run at about 10Hz, so 20 loop iterations is about 2 sec) */
      if(!avoidcount)
      {
        avoidcount = 15;
        randcount = 0;

        if(sp->GetScan (1) + sp->GetScan (15) < sp->GetScan (7) + sp->GetScan (8))
          newturnrate = -turnrate;
        else
          newturnrate = turnrate;

      }
      avoidcount--;
    }
    else
    {
      avoidcount = 0;
      newspeed = speed;

      /* update turnrate every 3 seconds */
      if(!randcount)
      {
```

```
            /* make random int tween -20 and 20 */
            //randint = (1+(int)(40.0*rand()/(RAND_MAX+1.0))) - 20;
            randint = rand() % 41 - 20;

            newturnrate = dtor(randint);
            randcount = 20;
          }
          randcount--;
        }

        // write commands to robot
        pp.SetSpeed(newspeed, newturnrate);
      }
    }
    catch (PlayerCc::PlayerError e)
    {
      std::cerr << e << std::endl;
      return -1;
    }
}
```

16. Selecione a aba robot1.cpp e compile o programa pelo menu (**Control Programs -> Compile Control Program**).

17. Selecione a aba robot2.cpp e compile o programa pelo menu (**Control Programs -> Compile Control Program**).

18. Se todos os passos acima derem certo, a simulação pode ser executada pelo menu (**Project -> Simulate**) e mostrará dois robôs se desviando de obstáculos, mostrados na Figura 5.11.

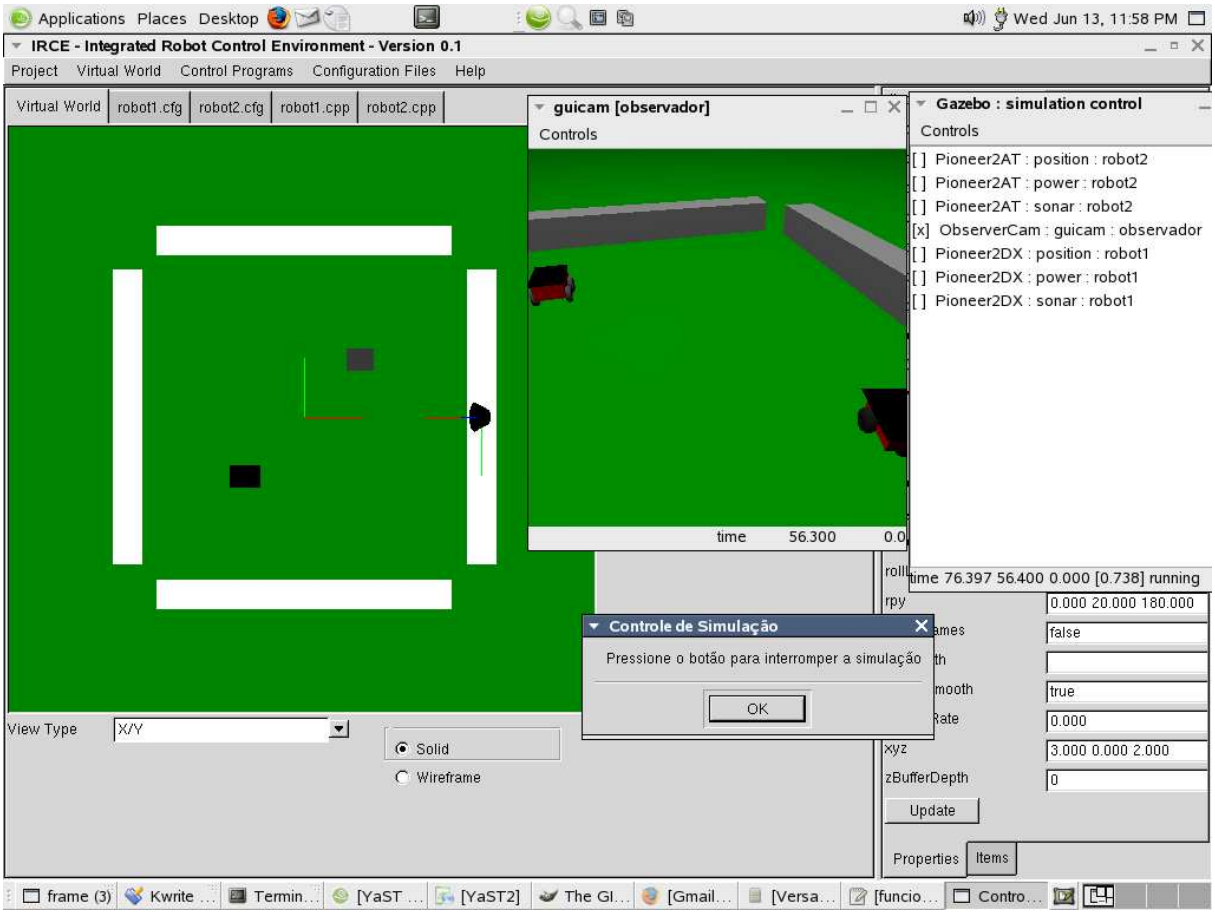


Figura 5.11: Execução da simulação do projeto exemplo.irce .

6 *Conclusões e Trabalhos Futuros*

Esta dissertação mostra um ambiente integrado de desenvolvimento de aplicações com robôs móveis que utiliza as ferramentas Player e Gazebo disponibilizadas pelo projeto P/S/G, que hoje é considerado o mais utilizado entre a comunidade de robótica que usa ferramentas de código-livre.

Embora reutilize código-fonte da ferramenta Gazebo, tal ambiente consiste numa aplicação isolada do projeto P/S/G que utiliza as ferramentas definidas nesse projeto como processos independentes para fazer a simulação do projeto.

Os seguintes requisitos foram implementados no ambiente:

- Gerência de Projeto IRCE;
- Montagem do Mundo Virtual;
- Configuração do Robô e seus Sensores;
- Programação de Controle;
- Simulação.

O protótipo do IRCE implementado é usável e pode ser utilizado pela comunidade robótica como um *software* de código-livre.

É importante notar que, por reutilizar código-fonte da ferramenta Gazebo para desenhar os modelos conforme mostrado na seção 4, o ambiente desenvolvido é sensível a alterações nas propriedades ou na geometria da classe C++ desses modelos.

Por outro lado, este trabalho introduz o conceito de desenvolvimento RAD (desenvolvimento rápido de aplicações) voltado para aplicações robóticas. Embora o desenvolvimento RAD exista há bastante tempo, ele ainda é pouco explorado para ajudar de forma realmente eficiente no desenvolvimento de controles de robôs especialmente em ambientes em 3D.

Existe também o aspecto da interatividade entre homem-máquina na construção de um mundo virtual. O IRCE pode ser estendido para incluir elementos inteligentes como a detecção de colisão entre os corpos que descrevem a cena ou a utilização de superfícies irregulares mais próximas do mundo real.

Pelo lado da simulação, além de robôs móveis ou manipuladores em terra, o Gazebo pode ser estendido para ser usado para aplicações envolvendo robôs aéreos e submarinos muito presentes em tarefas de vigilância e propecção de petróleo. Para isso, o novo simulador pode simular aspectos físicos tais como resistência do ar e de fluidos que são muito importantes para essas aplicações.

É importante notar que durante o desenvolvimento deste projeto, outros projetos relacionados à robótica evoluiram rapidamente. O próprio projeto P/S/G também amadureceu bastante e conta com uma comunidade cada vez maior de usuários.

Pode-se prever, num futuro muito breve, uma migracão na utilização de robôs móveis dos laboratórios para funções cada vez mais comuns no nosso dia-a-dia e essa ferramenta, assim como várias outras, pode contribuir para que isso aconteça.

Referências Bibliográficas

[1] BRUYNINCKX, H. Open robot control software: the orocos project. *Proceedings of the International Conference on Robotics and Automation*, v. 3, p. 2523–2528, 2001.

[2] THE Orocos Project. 2006. `http://www.orocos.org/`.

[3] SPECK, A. Robot simulation and monitoring on real controllers (robosim). *10Th European Symposium and Exibition*, p. 482–489, 1998.

[4] R., G. Object oriented programming for robotic manipulator simulation. *IEEE Robotics and Automation Magazine*, v. 4, n. 3, p. 21–29, september 1997.

[5] LOFFLER, M. S.; COSTESCU, N. P.; DAWSON, D. M. Qmotor 3.0 and the qmotor robotic toolkit: A pc-based. *IEEE Control Systems Magazine*, v. 22, p. 12–26, june 2002.

[6] GERKEY, B. P. et al. Most valuable player: a robot device server for distributed control. *Proceedings of the International Conference on Intelligent Robots and Systems*, v. 3, p. 1226–1231, 2001.

[7] THE Player StageProject. 2006. `http://playerstage.sourceforge.net`.

[8] MACDONALD, B. A.; COLLETT, T. H. J.; GERKEY, B. P. Player 2.0: Toward a practical robot programming framework. *Australasian Conference on Robotics and Automation (ACRA 2005)*, 2005.

[9] STAGE, 2D Multiple-Robo Simulator. 2006. `http://playerstage.sourceforge.net/index.php?src=stage`.

[10] MARCHEZI, H. C.; SCHNEEBELI, H. A. Um ambiente gráfico para desenvolvimento de controle para robôs móveis utilizando uma simulação 3d. *Anais do VIII Simpósio Brasileiro de Automação Inteligente*, p. 6, 2007.

[11] SILVA, E. L. d. S.; MENEZES, E. M. M. *Metodologia da Pesquisa e Elaboração de Dissertação*. Terceira. [S.l.]: UFSC/PPGEP/LED, 2001.

[12] JR, R. S. W.; SWEET, M. *OpenGL Super Bible Second Edition*. [S.l.]: Waite Group Press, 2000.

[13] OPEN Dynamics Engine. 2006. `http://www.ode.org/`.

[14] WXWIDGETS. 2006. `http://www.wxwidgets.org`.

[15] LIBXML2, The XML C parser and toolkit of Gnome. 2006. `http://xmlsoft.org/`.

[16] TINY XML, C++ XML Parser. 2006. `http://www.grinninglizard.com/tinyxml/`.

[17] RUMBAUGH J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. [S.l.]: Addison-Wesley, 1999.

[18] WOO, M. et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. [S.l.]: Addison-Wesley, 1997.

[19] PYRO, Python Robotics. 2007. `http://pyrorobotics.org/`.

[20] BLANK, D. S. et al. The pyro toolkit for ai and robotics. *AI Magazine*, AAAI Press, v. 27, n. 1, p. 39–50, 2006.

[21] GUMBLEY, L.; MACDONALD, B. A. Development of an integrated robotic programming environment. In: *Australasian Conference on Robotics and Automation*. Sydney: [s.n.], 2005.

[22] ECLIPSE.ORG. 2007. `http://www.eclipse.org/`.

[23] OLIVER, M. Webots tm: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, v. 1, n. 1, p. 40–43, 2004.

[24] CYBERBOTICS. 2006. `http://www.cyberbotics.com`.

[25] COCKBURN, A. *Writing Effective Use Cases*. [S.l.]: Addison-Wesley, 2000.

[26] STROUSTRUP, B. *The C++ Programming Language*. Third. AT&T Labs Murray Hill, New Jersey: Addison-Wesley, 1997.

[27] COAD, P.; YOURDON, E. *Object-Oriented Design*. [S.l.]: Yourdon Press, 1991.

[28] FOWLER, M. et al. *Patterns of Enterprise Application Architecture*. [S.l.]: Addison-Wesley, 2005. 379–386 p.

[29] STL Map. 2006. `http://www.sgi.com/tech/stl/Map.html`.

[30] STEPANOV, A.; LEE, M. *The Standard Template Library*. [S.l.]: Hewlett-Packard Company, 2005.

APÊNDICE A – Interfaces do Player

Interface (Classe C++)	**Descrição**
PlayerCc::ActArrayProxy	Controla um dispositivo actarray (array de atuadores)
PlayerCc::CameraProxy	Obtém imagens de um dispositivo de câmera
PlayerCc::DioProxy	Lê de um dispositivo dio (I/O digital)
PlayerCc::FiducialProxy	Controla um dispositivo fiducial
PlayerCc::GpsProxy	Controla um dispositivo de GPS
PlayerCc::GripperProxy	Usada para controlar um dispositivo gripper (garra)
PlayerCc::IrProxy	Controla um dispositivo IR
PlayerCc::LaserProxy	Controla um dispositivo de laser
PlayerCc::LocalizeProxy	Controla um dispositivo de localização que oferece hipóteses de várias posições e orientações para um robô
PlayerCc::MapProxy	Provê acesso a um dispositivo de mapa
PlayerCc::PlannerProxy	Provê acesso para um dispositivo planejador de movimento (motion planner) em 2D
PlayerCc::Position1dProxy	Controla um dispositivo position1d (de posicionamento unidimensional)
PlayerCc::Position2dProxy	Controla um dispositivo position2d (de posicionamento bidimensional)
PlayerCc::Position3dProxy	Controla um dispositivo position3d (de posicionamento tridimensional)
PlayerCc::PowerProxy	Controla um dispositivo de força
PlayerCc::RFIDProxy	Controla um dispositivo RFID (Dispositivo de Identificação por Radio Frequência)
PlayerCc::SimulationProxy	Provê acesso a um dispositivo de simulação (como o Stage ou Gazebo, por exemplo)
PlayerCc::SonarProxy	Controla um dispositivo de sonar
PlayerCc::SpeechProxy	Controla um dispositivo de fala
PlayerCc::SpeechRecognitionProxy	Provê acesso a um dispositivo de reconhecimento de fala
PlayerCc::WiFiProxy	Controla um dispositivo WiFi
PlayerCc::WSNProxy	Controla um dispositivo wsn

Tabela A.1: Algumas interfaces disponibilizadas pelo Player e a descrição dos dispositivos que representam.

APÊNDICE B – Drivers implementados para o Player

Driver	**Fabricante**	**Dispositivo**	**Interfaces providas pelo Driver**
p2os	MobileRobots	Robôs baseados nos sistemas operacionais PSOS/P2OS/AROS (exemplo: Pioneer e AmigoBot) e os acessórios integrados incluindo a CMUcam conectada a porta AUX.	*position2d "odometer"*: motor das rodas, *position2d "compass"*: bússola (se existir), *position2d "gyro"*: bússola giroscópica (se existir), *power*: voltagem da bateria atual, *sonar*: array de sonares (se existir), *aio*: porta analógica I/O (se existir), *dio*: porta digital I/O (se existir), *gripper*: garra (se existir), *actarray*: braço (se existir), *limb*: cinemática inversa do braço (se existir), *bumper*: array de bumper, *blobfinder*: CMUCam2 conectada a uma porta AUX em uma mesa P2OS (se existir), *sound*: sistema de som do AmigaBot, que pode tocar arquivos WAV
camera1394	Vários	Câmeras IEEE1394 (Firewire)	*câmera*
sonyevid30	Sony	Câmeras pan-tilt-zoom EVID30 e EVID100	*ptz*

Tabela B.1: Alguns exemplos de drivers e as interfaces que estes implementam no Player.

APÊNDICE C – Exemplo de programa controlador

Abaixo segue o exemplo de um código em C++ de um controlador que utiliza as bibliotecas disponibilizadas pelo Player. O programa abaixo implementa um algoritmo para desvio de obstáculos utilizando um robô móvel com rodas que possui um sensor de sonar.

```
#include <iostream>

// biblioteca disponibilizada pelo Player
#include <libplayerc++/playerc++.h>

int main(int argc, char *argv[])
{
  using namespace PlayerCc;

  // Conecta-se a um servidor player em localhost
  PlayerClient    robot("localhost");

  // Instancia um dispositivo sensor sonar
  SonarProxy      sp(&robot,0);

  // Instancia um dispositivo para controlar
  // as rodas do robô
  Position2dProxy pp(&robot,0);

  for(;;)
  {
    double turnrate, speed;

    // Le informação dos dispositivos
    robot.Read();

    // Imprime o conteúdo do sensor sonar na tela
    std::cout << sp << std::endl;

    // Implementa desvio simples de obstáculos
    if((sp[0] + sp[1]) < (sp[6] + sp[7]))
      turnrate = dtor(-20); // Vira 20 graus por segundo
    else
      turnrate = dtor(20);

    if(sp[3] < 0.500)
```

```
      speed = 0;
    else
      speed = 0.100;

    // Comanda os motores das rodas do robô
    pp.SetSpeed(speed, turnrate);
  }
}
```

APÊNDICE D – Exemplo de arquivo world

Abaixo segue um exemplo de arquivo world, cujo formato segue o padrão XML, utilizado pelo Gazebo para descrever o mundo virtual.

```
<?xml version="1.0"?>
<gz:world xmlns:gz="http://playerstage.sourceforge.net/gazebo/xmlschema/#gz"
 xmlns:model="http://playerstage.sourceforge.net/gazebo/xmlschema/#model"
  xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
   xmlns:window="http://playerstage.sourceforge.net/gazebo/xmlschema/#window"
    xmlns:param="http://playerstage.sourceforge.net/gazebo/xmlschema/#params"
     xmlns:ui="http://playerstage.sourceforge.net/gazebo/xmlschema/#params">

  <model:ObserverCam>
    <id>userCam0</id>
    <xyz>5.637 93.859 3.053</xyz>
    <rpy>-0 13 -39</rpy>
    <imageSize>640 480</imageSize>
    <updateRate>10</updateRate>
    <renderMethod>GLX</renderMethod>
  </model:ObserverCam>

  <model:LightSource>
    <id>light1</id>
    <xyz>0.000 0.000 100.000</xyz>
    <ambientColor>0.2, 0.2, 0.2</ambientColor>
    <diffuseColor>0.8, 0.8, 0.8</diffuseColor>
    <specularColor>0.2, 0.2, 0.2</specularColor>
    <attenuation>1.0, 0.0, 0.00</attenuation>
  </model:LightSource>

  <model:GroundPlane>
    <id>ground1</id>
    <color>0.0 0.5 0.0</color>
    <textureFile>grid.ppm</textureFile>
  </model:GroundPlane>

  <model:Pioneer2DX>
    <id>robot1</id>
    <xyz>8.647 91.267 0.200</xyz>

    <model:Pioneer2Gripper>
```

```
      <id>gripper1</id>
      <xyz>0.14003 0 -0.0975</xyz>
    </model:Pioneer2Gripper>

  </model:Pioneer2DX>

  <model:SimpleSolid>
    <xyz>10.0 91.8 0</xyz>
    <shape>sphere</shape>
    <size>.5</size>
    <fiducial>4</fiducial>
  </model:SimpleSolid>

</gz:world>
```