Uma ferramenta para o ensino de inteligência artificial usando jogos de computador

Filipe Correa Lima da Silva

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Flávio Soares Corrêa da Silva

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da Microsoft Research

São Paulo, Novembro de 2007

Uma ferramenta para o ensino de inteligência artificial usando jogos de computador

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Filipe Correa Lima da Silva
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Flávio Soares Corrêa da Silva (orientador) - IME-USP.
- Profa. Dra. Renata Wassermann - IME-USP.
- Prof. Dr. Esteban Clua - IC-UFF.

Agradecimentos

Agradeço primeiramente a DEUS, por ter me dado o dom da vida.

Agradeço aos meus queridos pais, Tânia Silva e Carlos Silva, os quais eu amo, dos quais tenho orgulho, e a quem dedico essa dissertação. Vocês foram a minha inspiração e contínua fonte de força para que eu pudesse ter terminado esse mestrado. Devo minha vida a vocês.

Agradeço ao meu querido tio e amigo Tad Correa, pessoa a qual amo, pela amizade, pelo companherismo e por ter me ensinado a persistir no caminho. Você é um exemplo para mim, porque persistiu em situações muito mais difíceis do que as que eu vivi durante esse trabalho.

Agradeço à minha querida avó, Maria José Corrêa, pessoa que amo, e que sempre acreditou desde o início que eu conseguiria terminar, mesmo acompanhando as dificuldades que tive no caminho.

Aos meus tios queridos, os quais amo, Jayro Guimarães e Amélia Guimarães, por terem me apoiado desde o início dessa jornada.

Agradeço ao meu orientador Flávio Soares Corrêa da Silva, que teve a idéia inicial do projeto, por ter acreditado no meu trabalho e por ter acreditado que eu conseguiria terminar o mestrado mesmo estando empregado. Lembro-me de diversas reuniões que tivemos, que mudaram totalmente minha visão do projeto e que me ajudaram a enxergar a direção a ser tomada em momentos em que me encontrava completamente perdido. Serei para sempre grato pela oportunidade que tive de estudar com o senhor.

Gostaria de agradecer ao Universo Online, meu empregador, por ter me apoiado durante essa jornada. Em especial, gostaria de agradecer a duas pessoas muito importantes que tiveram influência direta no meu trabalho. Ao Daniel Rodrigues Ambrósio, por ter acreditado em mim e por ter me ajudado a conseguir terminar o mestrado de diversas maneiras, gesto que nunca esquecerei.

Ao meu amigo, Cézar Monteiro Pirajá Neto, que trilhou uma parte dessa jornada comigo, e que

apesar da distância, é uma das pessoas importantes das quais lembrei quando busquei forças para terminar e que me motivou a seguir em frente em diversas conversas que tivemos ao longo do caminho.

Por fim agradeço à Microsoft Research, por ter me auxiliado financeiramente durante parte do desenvolvimento deste trabalho.

Resumo

A queda do interesse por parte de novos universitários, para cursos de ciência da computação em várias universidades do mundo [55, 68], é um sinal para começarmos a pensar se um dos motivos dessa queda tem relação com a forma pela qual o ensino de computação está sendo conduzido.

Nessa linha, perguntamos se existem maneiras de tornar o ensino de computação mais interativo e motivante para os alunos da nova geração, os quais cresceram no meio de uma das categorias mais complexas de software existentes hoje: os jogos de computador [10]. Esses softwares ficam cada vez mais interativos, complexos e ricos em detalhe com o passar do tempo.

Conforme será exposto, por meio do estudo de algumas iniciativas de pesquisadores nesse sentido, o ensino de ciência da computação pode se tornar mais interessante e rico com a utilização de jogos de computador como recurso didático, para capturar a atenção dessa nova geração de estudantes.

Com base nesse resultado, vamos focar nossa contribuição no ensino de lógica em cadeiras de Inteligência Artificial (IA), uma área de concentração da Ciência da Computação. Apresentamos uma ferramenta que chamamos de Odin, para construir e visualizar especificações executáveis de IA, por meio da linguagem PROLOG, em ambientes tridimensionais de jogos de computador.

Entendemos que essa ferramenta pode ser utilizada como um recurso didático em cursos de lógica para alunos em nível de graduação. Como principal benefício, o aluno tem a possibilidade de explorar, observar e interagir com os resultados de seu trabalho. Essa possibilidade de visualização é o que parece reter a atenção do aluno, de acordo com pesquisadores na área.

Disponibilizamos dois cenários de uso: O labirinto e o Mundo de Wumpus, dois cenários que juntos podem ser utilizados para cobrir uma boa parte da carga didática de um curso de lógica para graduação. Outros cenários podem ser desenvolvidos posteriormente por meio de extensão do *framework* composto por classes C++.

A ferramenta foi utilizada em duas cadeiras de inteligência artificial no Instituto de Matemática e Estatística, da Universidade de São Paulo. Consideramos que a recepção da ferramenta por parte dos alunos foi positiva.

Palavras-chave: ensino, lógica, PROLOG, visualização de programas.

Abstract

The interest in Computer Science related courses is dropping considerably as reported by various universities around the world [55, 68]. Among the factors that are causing this scenario to take place, one could certainly be the way Computer Science courses are being taught.

With that in mind, we have asked if there are ways to teach Computer Science in a more engaging and interactive manner, so as to appeal to the new generation of students that grew in constant touch with one of the most complex and sophisticated categories of existing software, the interactive 3D computer games [10]. This category of software is growing more intricate, complex and rich in detail day by day. And students seem to love it.

The literature review presented in the course of this work will show that the use of computer games in teaching, especially in computer science courses, can be an effective way of providing an engaging, interactive and visually rich learning environment for students of the new generation.

With this basis, our key contributions focus on the teaching of logic in Artificial Intelligence(AI) related courses, an important area within the Computer Science Curriculum. We propose a tool called Odin, for building executable specifications of AI, using the PROLOG language, in the form of interactive 3D computer games.

We understand that this tool can be used as a didactic resource for undergraduate logic courses. The main benefit is that it enables the student to explore, observe and interact with the result of his own designs, which is what seems to capture the students attention, according to researchers in the field.

We provide two scenarios: the labyrinth problem and the Wumpus World problem. Scenarios that combined can cover a large part of an undergraduate logic course Syllabus. The tool is extensible so that more scenarios can be added, by extending the framework composed of C++ classes.

The tool was used in two graduate AI courses, in the Statistic and Mathematics Institute of the University of São Paulo. We consider that the reception of the tool by the students was positive.

Keywords: teaching, logic, PROLOG, program visualization.

Sumário

Lista de Abreviaturas **xi**

Lista de Figuras **xiii**

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivo 2

1.3 Organização do Trabalho 2

2 Visualização no ensino e na pesquisa em Computação **5**

2.1 Jogos Interativos de Computador: a "Killer Application"da IA 5

2.2 Projeto Soar/Games 7

2.2.1 Soar 7

2.2.2 Objetivo 7

2.2.3 *AI Engine* 8

2.2.4 Resultados 8

2.3 M.U.P.P.E.T.S 9

2.3.1 Objetivo 9

2.3.2 Problemas no Ensino de Programação 10

2.3.3 Ambientes Virtuais Colaborativos 11
2.3.4 Funcionamento 15
2.3.5 Resultados 18
2.4 Alice 20
2.4.1 Objetivo 21
2.4.2 Funcionamento 22
2.4.3 Resultados 23
2.5 Considerações Finais 24

3 Tecnologias Utilizadas **27**
3.1 A linguagem PROLOG 27
3.2 Os Cenários de Aplicação 35
3.2.1 O Labirinto 35
3.2.2 O Mundo de Wumpus 36
3.3 *Engines* Gráficas 37
3.4 Integrando SWI-Prolog e C++ 40
3.5 A *Engine* de Renderização Gráfica OGRE 45

4 Modelagem da Ferramenta **51**
4.1 O Conjunto de Classes da Ferramenta 51
4.2 Cenário do Labirinto em Funcionamento 58
4.3 Cenário do Mundo de Wumpus em Funcionamento 62

5 Considerações Finais **71**
5.1 Contribuições 71
5.2 Uso da Ferramenta no IME/USP 72

5.3 Apresentação em Eventos 72
5.4 Trabalhos Futuros 72

A Apêndice **75**

A.1 Código da classe PrologAdapter 75
A.2 Programa PROLOG para o Cenário do Labirinto 81
A.2.1 Código Usando a Distância de Manhattan 81
A.2.2 Código Usando o Algoritmo A-estrela 83
A.3 Programa PROLOG para o Cenário do Mundo de Wumpus 90

Referências Bibliográficas **101**

Lista de Abreviaturas

IA	Inteligência Artificial.
API	*Application Programming Interface.*
LGPL	*GNU Lesser General Public License.*
OGRE	*Object-Oriented Graphics Rendering Engine.*
3D	Três Dimensões, Tridimensional.
2D	Duas Dimensões, Bidimensional.
BSP	*Binary Space Partition.*
UTC	*Unified Theory of Cognition.*

Lista de Figuras

2.1 Primeira visão do ambiente Muppets. O avatar do jogador é representado inicialmente como uma pequena esfera. [9]. 15

2.2 Ambiente Muppets com algumas janelas da IDE integrada. O aluno pode escrever seus programas sem sair do ambiente. [3]. 16

2.3 O console do Muppets aberto. [9]. 19

2.4 A arena do jogo Robocode implementado no Muppets. Vários tanques de guerra batalham entre sí. O tanque de guerra que prevalecer até o final é o vencedor. Cada aluno é responsável por escrever a lógica de seu tanque para competir com os colegas. [57]. 21

2.5 Interface do Alice. Os comandos à esquerda são arrastados para dentro da caixa de script, assim o aluno não precisa digita-los. Um botão play no canto superior esquerdo ativa a animação e o aluno pode ver os resultados de seu código. 23

3.1 O plano quadriculado. Aqui mostramos dois possíveis caminhos até o destino. Os quadrados em cinza são os obstáculos e o Agente está representado pela letra A. O caminho vermelho leva o agente até o destino apesar de o caminho azul ser uma escolha melhor. 36

3.2 Um possível cenário no mundo de Wumpus. O agente pode ser visto no canto inferior direito. O Wumpus se econtra logo dois quadrados acima. A letra C representa o odor exalado pelo Wumpus. Os quadrados com um furo no centro representam os precipícios e as letras B em suas adjacências, representam a brisa soprando. Por fim, o pequeno retângulo no centro do quadriculado representa a pepita de ouro. 38

3.3 Diagrama mostrando algumas das principais classes da *engine* OGRE. 47

4.1 Diagrama de classes da ferramenta. Chamamos a ferramenta pelo codinome *Odin*, que usamos também para definir o *namespace*. 52

4.2 A classe DiscreteGrid sendo renderizada pela OGRE. As linhas que representam a divisão entre os quadrados podem ser escondidas. A textura do plano como podemos ver no quadrado A, e a textura dos quadrados individuais como podemos ver no quadrado B, podem ser configuradas dinamicamente. 53

4.3 Cenário do Labirinto da ferramenta Odin. O agente é representado pelo robô. Os obstáculos, são representadas pelos discos amarelos. 60

4.4 O caminho válido escolhido pelo agente pode ser visualizado pelos pontos brilhantes no plano. 61

4.5 Nessa figura, o caminho escolhido pelo agente atravessa um obstáculo. 62

4.6 Nessa figura, os obstáculos foram criados à semelhança de um labirinto. Mesmo assim o agente consegue achar o caminho para o ponto escolhido. 63

4.7 O agente e o tabuleiro 4×4. Somente o quadrado no qual o agente se encontra está revelado. 65

4.8 Os sinais de perigo que podem ser percebidos em B, a brisa indicando que um abismo está próximo, e em A o odor representado pela substância esverdeada, que indica que o Wumpus está próximo. O agente se encontra sobre a pepita de ouro em C. Nesse momento, o agente já tem condições de deduzir qual é o quadrado onde o wumpus se encontra, em função dos quadrados visitados em que se pode perceber o odor. O Wumpus se encontra no quadrado indicado por D. 66

4.9 O tabuleiro revelado por completo. 67

Capítulo 1

Introdução

A queda do interesse de novos universitários por cursos de ciência da computação [55], é uma realidade preocupante em várias universidades no mundo. Isso naturalmente gera uma movimentação no sentido de buscar pontos que podem ser melhorados no ensino de ciência da computação com o objetivo de aumentar a retenção e a motivação de alunos.

A Inteligência Artificial (IA), é uma importante área no estudo de ciência da computação. Um dos objetivos fundamentais da IA, é implementar sistemas inteligentes que apresentem capacidades de um ser humano.

Uma aplicação emergente onde esse objetivo pode ser perseguido, e que vem chamando a atenção de vários pesquisadores recentemente [45, 67, 42, 37, 29, 47, 70, 11, 56, 19], são os ambientes interativos tridimensionais, como os jogos interativos de computador.

Nos últimos anos a IA tornou-se parte essencial desses jogos [29]. Na medida em que os jogos se tornam mais complexos e os consumidores exigem personagens e oponentes controlados por computador mais sofisticados, os programadores são obrigados a colocar maior ênfase no desenvolvimento da IA em seus jogos [67].

Assim, torna-se desejavel desenvolver ferramentas que permitam testar e utilizar técnicas de IA em jogos. Sob o ponto de vista de modelagem, como o ambiente virtual onde o jogo acontece pode ser tão complexo quanto se queira, implementar comportamento inteligente nos agentes que vivem nesse ambiente torna-se um problema complexo e interessante.

Atualmente, a disciplina de IA consta como obrigatória em muitos cursos de qualidade de graduação e pós-graduação em áreas ligadas à Tecnologia da Informação (TI). Dentre os cursos nos

quais isso não ocorre, tal disciplina consta como disciplina optativa regularmente solicitada. Este fato atesta a importância reconhecida de IA na formação do profissional moderno de TI e justifica uma atenção ao desenvolvimento de recursos didáticos eficazes e atraentes para o ensino de IA.

1.1 Motivação

No ensino de raciocínio formal e paradigmas lógicos em cursos de Ciência da computação, como Inteligência Artificial ou Lógica Formal, é desafiador encontrar exemplos de aplicação práticos e interessantes o suficiente para que os alunos da nova geração, que cresceram no meio de jogos de computador altamente avançados, se sintam motivados.

Nesses cursos, PROLOG é uma das linguagens mais tradicionalmente utilizadas. Por se tratar de uma linguagem de um paradigma completamente diferente do que os alunos estão acostumados ao longo do curso [13], na maioria das vezes linguagens Estruturadas ou Orientadas a Objeto, acreditamos que a visualização torna-se uma grande aliada para o aprendizado do aluno, tendo em vista os trabalhos que serão apresentados no capítulo 2.

1.2 Objetivo

Uma ferramenta que permita construir especificações executáveis de teorias de IA aplicada a jogos, pode ser usada em disciplinas de IA para o desenvolvimento de projetos didáticos estimulantes e atraentes para os alunos, que podem apreciar os resultados de seus trabalhos como protótipos de jogos por eles desenvolvidos.

O objetivo no presente trabalho, é estudar iniciativas no uso de jogos de computador em pesquisa e ensino de computação, e com essa base, contribuir para o ensino de Inteligência Artificial por meio da disponibilização de uma ferramenta que chamamos de Odin, para construir especificações executáveis de IA para jogos de computador, voltada primordialmente para uso como recurso didático no ensino de IA para o nível de graduação e utilizando a linguagem PROLOG, tradicionalmente utilizada nestes cursos.

1.3 Organização do Trabalho

O trabalho está organizado da seguinte maneira: no capítulo 2 discutimos o uso de IA em jogos, o uso de jogos no ensino de computação em geral e o uso de jogos no ensino de IA, apresentando trabalhos bem sucedidos nessa linha.

No capítulo 3 apresentamos as tecnologias e decisões de projeto que nortearam o desenvolvimento da ferramenta de apoio ao ensino de IA. No capítulo 4 apresentamos o conjunto base de classes da ferramenta, o funcionamento dos cenários de aplicação e exemplos de utilização da ferramenta com códigos em PROLOG.

Finalmente, os comentários sobre possíveis extensões da ferramenta, bem como demais considerações finais são encontradas no capítulo 5.

Capítulo 2

Visualização no ensino e na pesquisa em Computação

A inteligência artificial pode ser entendida como o estudo de agentes que existem em um ambiente e que podem agir e receber estímulos desse ambiente [62]. Com ambientes interativos tridimensionais, tais como jogos de computador, temos efetivamente um meio onde podemos criar esses ambientes e populá-los com agentes que podem receber informações sensoriais e agir. Nesse capítulo vamos expor alguns trabalhos que utilizam e propõem estes ambientes no ensino de ciência da computação e na pesquisa em Inteligência Artificial.

2.1 Jogos Interativos de Computador: a "Killer Application"da IA

No trabalho de Laird e van Lent [45], encontramos que nos últimos 30 anos, a IA foi se fragmentando em campos mais especializados, com enfoque em problemas mais específicos e utilizando algoritmos mais e mais especializados para resolvê-los. Por outro lado, tem ocorrido pouco progresso na construção de sistemas que se aproximem da inteligência humana, ou utilizando o termo inglês, "Human-Level AI".

Os autores definem sistemas de "Human-Level AI", como aqueles com os quais nós sonhamos quando vemos os robôs C3PO e R2D2 no filme *Star Wars*, ou HAL em *2001, A space Odyssey*. Eles apresentam muitas características de inteligência humana: resposta em tempo-real, robustez, interação inteligente autônoma com o ambiente, planejamento, comunicação em linguagem natural, emoções, raciocínio, senso comum, criatividade e aprendizagem.

Os autores dizem que jogos interativos de computador são a "*Killer Application*" para a pesquisa em IA. *Killer application* é um termo em inglês que não tem tradução direta. Uma *killer application* é um programa de computador tão útil ou tão desejável, que acaba por provar o valor de uma tecnologia

subjacente. O autor está sugerindo em outras palavras, que se a pesquisa em IA for aprofundada e aplicada com sucesso nos jogos de computador, então a procura por jogos vai aumentar muito, provando o valor da tecnologia subjacente que atua como diferencial, a IA.

Os autores apresentam uma lista de razões para os pesquisadores em IA levarem a indústria de jogos de computadores a sério:

- Primeiro, os desenvolvedores de jogos estão começando a reconhecer a necessidade de se construir personagens mais inteligentes. É interessante notar que o trabalho de Laird e Lent [45] foi publicado em 2000. Podemos encontrar em [32] que a indústria reconhece a necessidade e está ativamente trabalhando para construir personagens cada vez mais inteligentes.

- Segundo, a indústria de jogos é altamente competitiva e um componente forte dessa competição é a tecnologia. Uma das tecnologias mencionadas como diferencial de sucesso para os futuros jogos é a IA.

- Terceiro, em 2003 *Programador de IA* já é um cargo comum na indústria [14].

- Quarto, em termos de receita bruta, a indústria de jogos é maior do que a indústria do cinema.

- Quinto, a tendência de executar o processo de renderização por inteiro nas placas gráficas libera o processador, o que significa que podemos esperar mais processamento para os algoritmos de IA à medida em que o hardware evolui.

- Sexto, a indústria de jogos precisa da IA acadêmica. A ênfase atual na IA dos jogos é dar a ilusão de comportamento humano para situações limitadas. Na medida em que os jogos ficam mais realistas em termos de física e gráficos, é de se esperar que a construção de personagens mais inteligentes seja o próximo passo em direção ao realismo. Como pesquisadores, podemos utilizar estes ambientes cada vez mais realistas para a avançar na construção de agentes cada vez mais inteligentes.

Finalmente, cabe acrescentar que o ambiente onde o jogo acontece é sim virtual, porém não é uma simulação do domínio do problema: ele é o próprio domínio do problema [5].

2.2 Projeto Soar/Games

John E. Laird foi um dos principais responsáveis pela construção da arquitetura Soar [43], juntamente com Allen Newell e Paul Rosenbloom. O projeto Soar/Games [44], liderado por Laird, utiliza jogos criados com a arquitetura Soar para a pesquisa em IA.

2.2.1 Soar

A Soar é o resultado de um projeto que teve como objetivo a construção do que Alan Newell chama de teoria unificada de cognição (*unified theory of cognition - UTC*) [53].

A ciência cognitiva é uma disciplina que busca integrar expertise de várias outras disciplinas que de alguma forma estão relacionadas com algum processo cognitivo. Como exemplos, podemos citar lingüística, psicologia, inteligência artificial e antropologia. Cada disciplina tem sua maneira própria de observar e explicar os seus fenômenos cognitivos de interesse.

Newell via um problema nesse contexto, que ele resumiu no que ele chamava de microteorias, o que basicamente significa que cada disciplina explica um fenômeno cognitivo utilizando um subconjunto de princípios de uma teoria maior. Cada um desses subconjuntos seria uma microteoria.

Newell acreditava na hipótese de que as microteorias estavam relacionadas entre sí porque todo comportamento inteligente é gerado por apenas um sistema que é a mente. Cada microteoria estava explicando um comportamento ou fenômeno cognitivo que ultimamente é produzido pelo mesmo conjunto de mecanismos e princípios.

Newell propõe então integrar esses resultados para tentar formular uma teoria que unifica as microteorias, o que ele vai chamar de Teoria Unificada de Cognição ou UTC. O objetivo de Newell é representar essa teoria por meio de uma arquitetura computacional, e assim surgiu a Soar.

No começo do projeto SOAR significava *State, Operator And Result*. Atualmente, Soar tornou-se apenas o nome da arquitetura e não é mais utilizado como um acrônimo. O projeto começou em 1987 e envolveu diversas universidades. Uma introdução à arquitetura Soar pode ser encontrada em [46].

2.2.2 Objetivo

O objetivo do projeto Soar/Games é aplicar técnicas do estado da arte de IA a jogos de computadores por meio do desenvolvimento de agentes inteligentes [44]. Utilizando a arquitetura Soar,

foram desenvolvidos agentes que planejam e aprendem e foram desenvolvidos ambientes que servem como base para testar resultados de pesquisa em aprendizagem de máquina, arquiteturas inteligentes e projetos de interface.

2.2.3 *AI Engine*

Um aspecto central do projeto foi o de construir um *AI Engine* que pudesse ser reutilizado em diversos jogos, com o objetivo de reduzir o tempo de desenvolvimento. O desenvolvedor precisaria se preocupar apenas com a adaptação da *engine* às particularidades de cada jogo.

Laird divide a *AI Engine* em três componentes: a máquina de inferência, a base de conhecimento, e a interface com o jogo.

O objetivo da *máquina de inferência* é aplicar o conhecimento do agente à situação atual. A situação atual do agente é representada por estruturas de dados e informação contextual. A máquina de inferência está constantemente operando em um ciclo de decisão: perceber, pensar e agir.

A *interface*, o segundo componente da *AI Engine*, é o canal de comunicação da máquina de inferência com o ambiente virtual. A interface tem a responsabilidade de extrair informações sensoriais do ambiente e alimentar a máquina de inferência, bem como comunicar ao ambiente as ações escolhidas pela máquina de inferência.

Como os jogos podem variar bastante em termos do ambiente virtual em que se inserem, cada *AI Engine* vai precisar de uma interface diferente. A interface deve disponibilizar para a máquina de inferência exatamente as mesmas informações sensoriais que um jogador humano recebe. Laird diz que a interface deve acessar a estrutura de dados de jogo diretamente, evitando assim todos os problemas de visão computacional envolvidos.

O último elemento de um *AI Engine* é a base de conhecimento. Como cada jogo é diferente, é impossível pensar em reaproveitar totalmente a base de conhecimento. Mas um motor de IA pode disponibilizar uma base de conhecimento geral para um gênero específico de jogos. Ela seria composta de objetivos, táticas e comportamentos independentes do jogo.

2.2.4 Resultados

Como resultado do Projeto Soar/Games, Laird cita experimentos com 2 jogos comerciais: *Quake2* e *Descent3*. No jogo *Quake2*, foi implementado um agente chamado quakebot, que tem por objetivo

jogar quake tão bem quanto um humano. O quakebot derrota facilmente iniciantes e é um forte desafio para jogadores mais experientes. O agente implementado para o jogo *Descent3* consegue explorar o jogo derrotando os monstros existentes. Adicionalmente, foram feitos experimentos utilizando o sisema KnoMic (*Knowledge Mimic*) [66], onde o agente quakebot foi capaz de aprender assistindo um jogador experiente jogar *Quake2*.

Outro trabalho recente do projeto Soar/Games, é o desenvolvimento de um jogo de aventura onde agentes inteligentes fazem diferença. O jogo está sendo implementado como uma extensão do *Unreal Tournament* e é chamado de *Haunt2*.

O jogador controla um fantasma, trazido de sua dimensão por um cientista malvado que o aprisiona em uma casa em que habitam personagens que são agentes inteligentes. O foco da aventura, está na interação do jogador com os agentes.

O objetivo do jogo é conseguir uma forma de fazer o fantasma voltar para a dimensão em que vivia, usando para isso poderes especiais para influenciar os agentes e tendo como desafio contornar as limitações impostas a um ser incorpóreo, como por exemplo, não poder pegar ou mover objetos. [6, 47, 42].

Laird cita em [67], que jogos são meios acessíveis, orientados a ação e têm um forte apelo visual. Esses fatores levam a uma interessante demonstração de pesquisa aplicada em inteligência artificial.

2.3 M.U.P.P.E.T.S

Muppets é um acrônimo que significa *"Multi User Programming Pedagogy for Enhancing Traditional Study"* ou algo como *Pedagogia de Programação Multi Usuário para Enriquecimento do Estudo Tradicional*. Muppets é um projeto de software que foi iniciado em meados de 2002 no Rochester Institute of Technology e é liderado pelo professor Andrew M. Phelps [3].

2.3.1 Objetivo

O objetivo do projeto Muppets é atenuar a difícil experiência dos calouros em cursos voltados para programação de computadores. O projeto Muppets procura atingir esse objetivo fomentando o engajamento de alunos veteranos no ensino de programação por meio de uma ferramenta interativa tridimensional que funciona como um Ambiente Virtual Colaborativo.

2.3.2 Problemas no Ensino de Programação

É sabido que existem dificuldades intrínsecas em relação ao aprendizado de programação [35]. Programar é uma tarefa que exige extrema precisão: uma vírgula é a diferença entre um programa que é aceito pelo computador e outro que não. Uma simples ordem de execução diferente entre dois comandos pode ser a diferença entre um programa que funciona corretamente e outro que pode causar prejuízos enormes para a companhia e para seus clientes.

Além disso, programar significa solucionar de maneira intensa problemas de lógica e matemática como afirma o famoso pesquisador Dijkstra em [1]. São problemas de lógica, matemática e arquitetura de sistemas que exigem que o aluno tenha que alavancar muitas habilidades diferentes, as quais muitas vezes não desenvolveu completamente, para que ele possa ter um retorno mínimo em termos de aprendizado e produtividade.

A imagem dos cursos intensivos em programação também é negativa. Tony Jenkins alega em [35] que esses cursos têm a reputação de serem muito difíceis mesmo que por vezes essa percepção seja exagerada. Aliado a isso, a imagem do programador de computadores é negativa também, pois são percebidos como *Geeks*, termo inglês que significa uma pessoa excessivamente interessada por tecnologia com diversas aspirações intelectuais e que por vezes apresenta vivência social abaixo da média. O autor conclui essa constatação dizendo que o prospecto de um curso difícil, aliado com a imagem negativa que apresentam as pessoas que o completam com sucesso, tem um impacto significativo no rendimento do aluno no curso.

Outro problema é a motivação que os alunos de hoje apresentam para aprender programação. Em [34] encontramos uma pesquisa feita nas universidades de Leeds e Kent no Reino Unido, sobre a motivação dos estudantes em cursos que contém programação de computadores em seus currículos.

Um dos resultados da pesquisa mostra que apenas 20% dos alunos estão motivados em aprender a programar quando perguntados a respeito de sua motivação na escolha de um módulo de programação em seus cursos. Quase 50% dos alunos entrevistados alegaram que o módulo de programação era apenas uma etapa obrigatória a ser completada em direção ao objetivo final que é a obtenção do diploma.

O autor conclui dizendo que os professores de programação precisam levar em conta essas diferentes motivações para ensinar programação de maneira efetiva. Um aluno que está motivado a aprender programação, ao contrário do aluno que está fazendo a disciplina apenas para conseguir os

créditos, irá além do que é dado na sala de aula e terá um aprendizado mais efetivo.

O projeto Muppets procura responder a pergunta de como mudar as motivações dos alunos para que eles se interessem genuinamente em aprender programação. *Ambiente Virtual Colaborativo* é um dos conceitos chave que o projeto Muppets utiliza para responder essa pergunta.

2.3.3 Ambientes Virtuais Colaborativos

Ambientes Virtuais Colaborativos vêm do termo em inglês CVE, ou Collaborative Virtual Environments. O estudo de CVEs é uma sub-área dentro da Realidade Virtual. O objetivo do estudo em CVEs, segundo Churchill, Snowdon e Munro [15], é "*prover meios inovadores e mais eficazes de se lidar com os computadores como um meio de comunicação e compartilhamento de informações*".

Bouras, Triantafillou e Tsiatsos definem CVEs da seguinte maneira [12]: um simples *Ambiente Virtual* é um sistema computacional, que gera um ambiente virtual tridimensional, com o qual o usuário pode interagir e receber feedback em tempo real. Se vários usuários podem se conectar nesse sistema, então dizemos que se trata de um *Ambiente Virtual Compartilhado* ou *Ambiente Virtual Multi Usuário*. Um CVE é então, um *Ambiente Virtual Compartilhado* direcionado a uma tarefa colaborativa. A questão de o ambiente ser tridimensional não é necessariamente um requisito [59].

Tanto que uma das primeiras formas de Ambiente Virtual Colaborativo, foram os MUDs, um acrônimo em inglês para *Multi User Dungeon* que não tem realmente uma tradução direta. O jogo é baseado em interfaces de texto. O termo *Dungeon* vem do mundialmente famoso jogo de mesa *Dungeons and Dragons* (D&D) (ou "Calabouços e Dragões" em tradução direta) que surgiu por volta de 1974 [54]. Aqui vale uma explicação do conceito de D&D.

D&D é um RPG (Role Playing Game) ou um jogo onde os jogadores devem assumir o papel de um personagem e imaginar estarem inseridos em uma cenário com certa ambientação. No caso de D&D o cenário se passa em mundos medievais onde os jogadores podem assumir papéis de guerreiros brandindo espadas, magos conjurando magias ou clérigos que têm o poder de curar e ressucitar outros personagens e entidades.

Para um jogo de D&D acontecer, devem estar presentes pelo menos duas pessoas: um mestre (Dungeon Master) e um jogador. O mestre é responsável por criar e contar a história. Ele é responsável por todos os acontecimentos no mundo que não são originados pelos jogadores. Ele cria e mantém o mundo no qual os jogadores se aventurarão ao longo das sessões de RPG.

No D&D os jogadores devem interpretar seus personagens imaginando que estão vivendo dentro do cenário naquele momento, essencialmente assumindo outra identidade. Nesse sentido é como se fosse um teatro. Diferentemente do teatro, ninguém sabe que final o jogo vai ter porque os jogadores podem fazer o que quiserem e não existe um roteiro. A história é criada dinâmicamente de acordo com as ações dos jogadores.

Ao longo das sessões, os jogadores ganham pontos de experiência quando fazem uma boa interpretação de seus personagens e quando derrotam monstros ou adversários em geral. Quanto maior o *nível* de um inimigo, mais pontos de experiência os jogadores obtem. Quando os jogadores acumulam uma certa quantidade de pontos, eles atingem o próximo nível. Pode-se dizer por exemplo "Tenho um guerreiro de nível 3". Ao subir de nível, o jogador adquire novos poderes e habilidades. Pontos de experiência essencialmente significam poder.

Geralmente, a maior parte dos pontos de experiência ganhos pelos jogadores vêm dos *Dungeons*. Um *Dungeon* é um lugar normalmente fechado onde os jogadores devem se aventurar para atingir algum objetivo, como por exemplo, resgatar algum objeto valioso, derrotar algum inimigo ou conseguir uma informação crítica. Um *Dungeon* pode ser um castelo, ou uma masmorra e normalmente é linear: os jogadores têm que trabalhar em conjunto para vencer uma série de obstáculos para chegar no final do Dungeon.

Ao longo do tempo, o termo *Dungeon* se tornou um conceito no gênero de RPGs. Essencialmente, é um lugar onde os jogadores devem cooperar e interagir para vencer uma série de obstáculos enquanto acumulam poder.

Em meados de 1978, Richard Bartle e Roy Trubshaw da universidade de Essex na Inglaterra, criaram o primeiro MUD do mundo. Ele era conhecido pelo título *British Legends* ou Lendas Britânicas e rodava nos servidores da universidade. O jogo, assim como seus sucessores, tinha diversos elementos do D&D. Diferentemente do D&D, MUD era jogado no computador, por meio de uma interface de texto. Outra diferença, é que não existia um mestre assim como no D&D, mas sim vários jogadores que iam moldando o mundo ao decorrer do tempo.

O jogo fez grande sucesso rapidamente. De fato, não era incomum a tradução do Acrônimo MUD significar *Multiple Undergraduate Destroyer*, significando algo como Multi Destruidor de Graduandos, alertando para o fato de que os graduandos despendiam quantidades enormes de tempo durante a faculdade para jogar *British Legends*. Isso é um exemplo do poder que esse tipo de jogo possui de capturar a atenção do jogador. Existem trabalhos que apontam elementos que podem explicar esse

fenômeno.

Em seu livro *A Comunidade Virtual* [60], Howard Rheingold fala sobre identidades virtuais alternativas e explica como é poderoso o grau de ligação que o jogador possui com seu personagem. Os relacionamentos virtuais que são formados nesses tipos de jogos por vezes têm ligações até mesmo mais fortes do que relacionamentos no mundo real.

Muitos Dungeons nos MUDs são muito difíceis para jogadores enfrentarem sozinhos e isso cria diversas situações para colaboração e interação entre os jogadadores. Você consegue expressar emoções em um MUD utilizando os comandos corretos. Rheingold explica que isso cria riqueza na comunicação, que faz da comunidade virtual um meio único de interatividade e que prende a atenção do jogador.

A identidade virtual em um jogo online desse tipo é muito importante para o jogador. Tão importante que o jogador fica emocionalmente conectado ao seu personagem ao ponto de causar pequenos abalos emocionais em certas situações. Por exemplo, em alguns MUDs, se um personagem morre, não existe nenhuma maneira de recupera-lo. A perda de um personagem tem um impacto muito forte na mente de um jogador. Howard Rheingold cita uma conversa que ele teve com o criador do *British Legends*, Richard Bartle [60]:

> *Perder seu personagem em um jogo é algo absolutamente terrível. É a pior coisa que pode acontecer com você e as pessoas ficam realmente chateadas... Não é como dizer "Que pena, perdi meu personagem" da mesma maneira que você diz "Que pena, perdi meu sapato". Não é nem mesmo como dizer "Que pena perdi meu personagem" do mesmo jeito que se diz "Que pena, perdi meu hamster de estimação". Está mais para "Eu morri!. Eles me mataram!". Não é como dizer "Eu acabei de perder todo aquele tempo e esforço que eu coloquei no meu personagem". A sensação está mais para "Eu acabei de morrer, isso é terrível! ó meu Deus, estou morto, vazio".*

Existe a possibilidade de se criar objetos virtuais em um MUD. Por exemplo pode-se criar um livro contendo uma autobiografia do seu personagem que pode ser lida por todos os outros jogadores. Criar um objeto que outros jogadores podem usar ou manipular provoca um poderoso efeito no jogador. O Muppets incorpora diretamente esse resultado em suas características com o objetivo de encorajar o aluno a criar objetos Muppets como veremos na próxima seção. O próprio autor Howard Rheingold, que jogava enquanto pesquisava comunidades virtuais, descreve a sensação de criar um

objeto que outros jogadores podem ver e usar:

> *"Uma das maiores recompensas em um MUD social vem da criação de uma ferramenta, objeto ou algo que cause uma pequena surpresa, que outros jogadores vão querer usar, comprar ou copiar."*

O termo *multi usuário* nos leva à importante questão da audiência. Diferentemente do D&D, onde você pode interagir com diversas pessoas em volta de uma mesa ou tabuleiro, nos MUDs você interage com milhares de pessoas conectadas no jogo de diferentes partes do mundo. Esse fato tem implicações importantes nos conceitos de glória e vergonha. Jonathan Baron explica em [7] o poder da glória e da vergonha:

> "Glória e vergonha são conceitos tão poderosos que deixaram culturas unidas por séculos, motivaram inúmeros indivíduos a arriscar suas vidas, e levaram muitas pessoas a acabarem com suas vidas."

Não pode existir glória ou vergonha sem que exista uma audiência, e nos MUDs ou MMORPGs de hoje essa audiência vem aos milhares. MMORPGs vem de *Massively Multi-Player Online Role Playing Games* ou RPGs massivamente Multi-usuário. Enquanto os MUDs eram baseados em interfaces de texto, as interfaces dos novos jogos online são tridimensionais e muito rica em detalhes. A audiência nesses jogos pode chegar a números impressionantes. Em janeiro de 2007, o jogo *World of Warcraft* atingiu a marca de 8 milhões e meio de jogadores no mundo.

Essa audiência tem uma característica especial. Baron explica que é como se você fosse um jogador de futebol profissional e sempre que fosse jogar, o estádio estaria cheio de torcedores que são jogadores profissionais também. Além disso, a partir do seu primeiro contato com o futebol, quando você ainda não conhece nehum fundamento do esporte, você tem uma grande torcida assistindo a cada movimento seu constantemente.

Esse conceito de glória e vergonha, que podem se manifestar como formas de incentivo e pressão social dependendo do jogo, é explorado pelo Muppets para fazer com que os alunos veteranos se vejam motivados a interagir com os calouros no ensino de computação por meio do uso da ferramenta como veremos na próxima seção.

2.3.4 Funcionamento

Nessa seção veremos o funcionamento do Muppets e como as funcionalidades do sistema se relacionam com os conceitos apresentados na seção 2.3.3.

O Muppets é um ambiente virtual tridimensional compartilhado e apresenta três grupos de interface: a interface tridimensional, o console e a IDE [9]. Ao entrar no Muppets, o aluno se depara com a interface tridimensional, como mostra a figura 2.1.

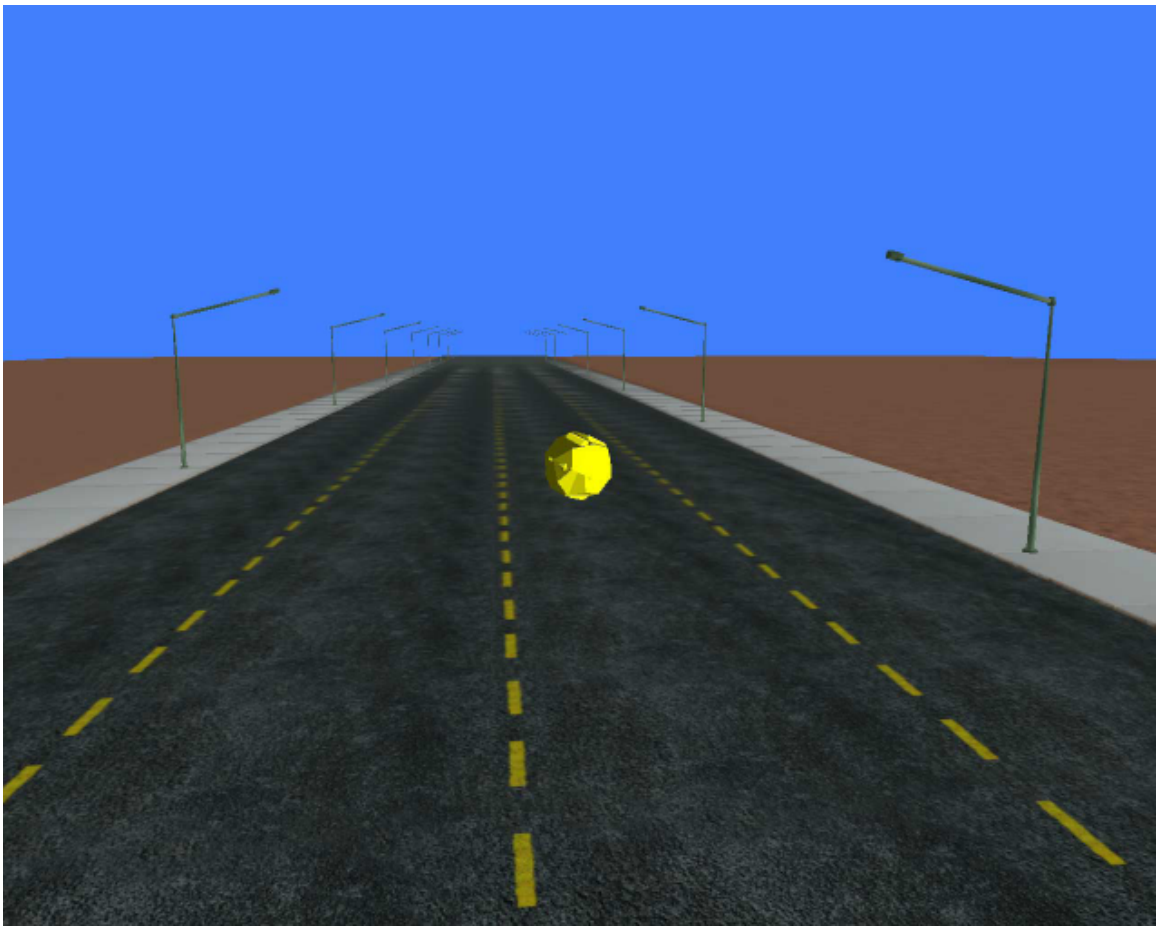

Figura 2.1: Primeira visão do ambiente Muppets. O avatar do jogador é representado inicialmente como uma pequena esfera. [9].

A figura mostra uma esfera no centro da tela que é a representação inicial do *avatar* do aluno. Um

avatar é uma representação virtual da pessoa em um ambiente virtual. O aluno pode se mover pelo ambiente com seu avatar e interagir com objetos que encontre pelo caminho ou até mesmo outros alunos que estejam no ambiente.

O Muppets possui uma IDE - *Integrated Development Environment*, ou ambiente integrado de desenvolvimento, que pode ser utilizada dentro do ambiente tridimensional. Dessa maneira, o aluno não precisa sair do ambiente para escrever seu código. A IDE do Muppets oferece as funcionalidades básicas de um editor de código como *syntax highlighting* e *code completion* e pode ser vista na figura 2.2

Figura 2.2: Ambiente Muppets com algumas janelas da IDE integrada. O aluno pode escrever seus programas sem sair do ambiente. [3].

Por meio da IDE o aluno pode escrever objetos Muppets. Objetos Muppets são classes Java que

implementam a interface `MuppetsObject`. O aluno pode criar objetos de complexidade arbitrária se souber sobrescrever os métodos corretos dessa interface. Por padrão, o modelo que é renderizado é um cubo. A listagem de código abaixo, ilustra uma possível implementação de um cubo que muda de cor ao longo do tempo [56].

```
import java.util.Random;
import java.lang.Math;
public class ColorCube extends MuppetsObject {

  private float clock; //um temporizador
  private float[] color; //vetor de float que guarda a cor

  //construtor
  public ColorCube() {
    Random rand;
    rand = new Random();
    color = new float[3];
    clock = rand.nextFloat() * 6.284f;
  }

  //sobrescrevendo o metodo para definir a animacao de cor
  public void update(float dt) {
    clock += dt; //atualiza o temporizador com o intervalo

    //criando a cor
    color[0] = ((float) Math.sin(clock) + 1) / 2.0f;
    color[1] = ((float) Math.cos(clock) + 1) / 2.0f;
    color[2] = ((float)Math.sin(clock)*(float)Math.cos(clock)+1)/2.0f;


    //o cubo assume a cor criada
    setColor(color[0], color[1], color[2]);
```

```
  }
}
```

Tudo o que o aluno teve que fazer nesse código, foi criar uma maneira de variar a cor do cubo em função do parâmetro `dt` do método `update`, que contém a fração de segundos que se passou desde a renderização do quadro anterior. Depois que o aluno escreve esse código e o compila, ele pode instanciar vários objetos do tipo `ColorCube`, bastando que para isso ele use o console.

Podemos ver o console ativo na figura 2.3. Pelo console, o aluno consegue enviar diversos comandos para o Muppets, sendo que na maioria das vezes o aluno vai usar o console para criar e publicar objetos. O aluno simplesmente digita `new ColorCube` no console e o novo cubo aparece no campo de visão do seu avatar, mudando de cor.

Como o Muppets é um ambiente virtual compartilhado, os alunos podem ver e interagir com seus respectivos avatares, além de poderem conversar enviando mensagens instantâneas entre sí. Se um aluno instancia um objeto, todos os outros alunos que estão nas proximidades nesse momento também vêem o objeto. Ainda, se o aluno escolher, ele pode publicar esse objeto para que outros colegas possam instanciá-lo por meio do console.

Estas funcionalidades estão baseadas nos conceitos apresentados na seção 2.3.3. A idéia é que os alunos aprendam a programar enquanto experimentam essas possibilidades do ambiente, com a alavancagem de todos os fatores psicológicos e sociais inerentes de uma CVE desse gênero.

O Muppets foi desenvolvido na linguagem Java utilizando uma complexa integração com C++ por meio da interface JNI para as chamadas de baixo nível que controlam a parte gráfica. Um artigo com detalhes técnicos sobre esse assunto pode ser encontrado em [58]. Recentemente, foi desenvolvido um módulo para suportar a linguagem C# da plataforma .NET da Microsoft [3], então os alunos podem aprender utilizando Java ou C#.

2.3.5 Resultados

Em 2007, o Muppets se encontra em estágio *alpha* de desenvolvimento [3] e um dos experimentos com o ambiente pode ser encontrado em [9]. Nesse experimento, o jogo Robocode foi implementado.

O RIT experimentou em um dos cursos de programação utilizar o jogo Robocode como uma ferramenta didática. O Robocode é uma arena onde vários tanques de guerra batalham entre sí. O tanque que prevalecer até o final será o vencedor. Os participantes são responsáveis por escrever a

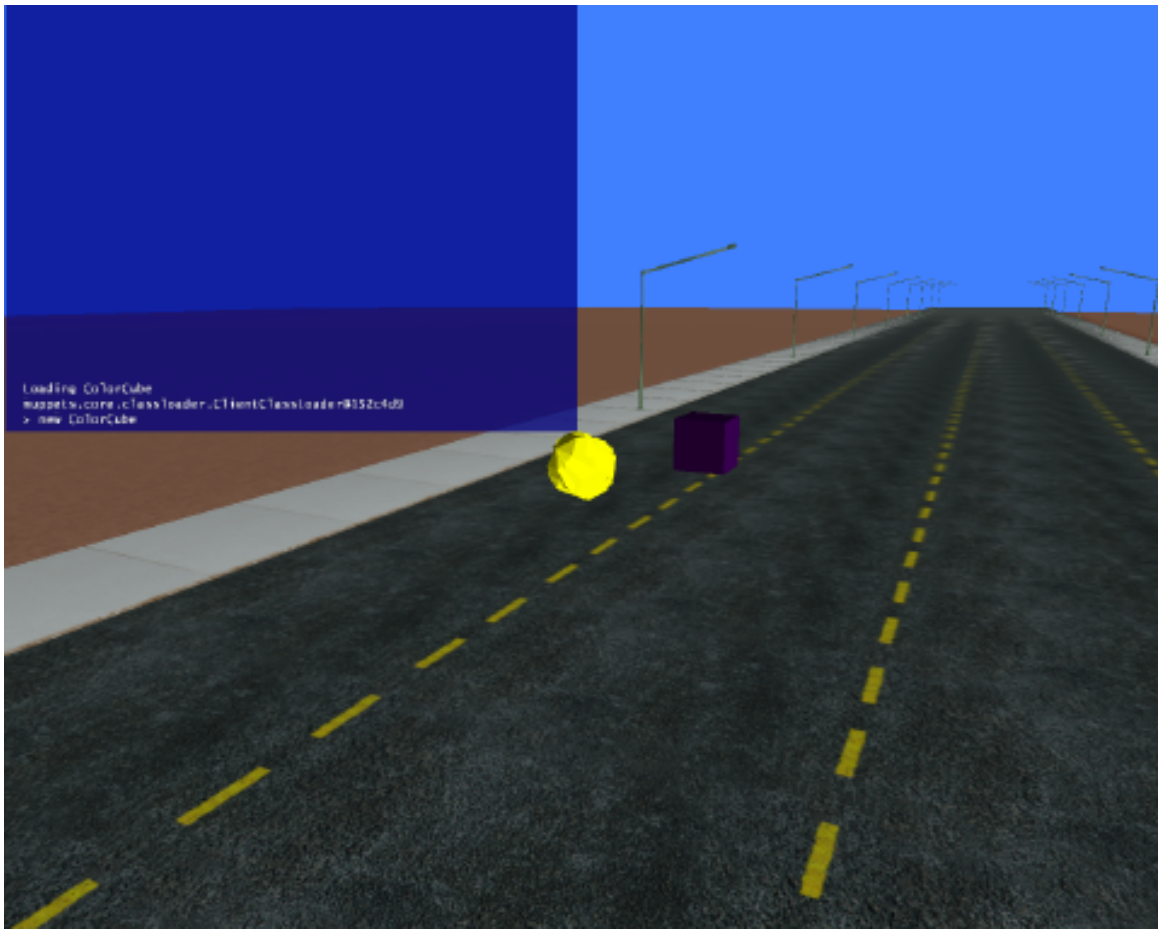

Figura 2.3: O console do Muppets aberto. [9].

lógica do tanque em Java e o Robocode faz a simulação da batalha utilizando esse código.

O experimento com o Robocode teve sucesso no início, porque foi descoberto que os alunos gostavam de ver a lógica de seus programas em funcionamento. Esse é um ponto importante, e é uma característica comum entre os projetos citados nesse capítulo. Phelps, Bierre e Parks constatam esse fato depois de compararem diferentes ambientes voltados para programadores iniciantes:

> "A característica principal que esses projetos têm em comum é possibilitar que os alunos vejam seus programas em funcionamento. Essa característica de poder fazer com que um determinado efeito ocorra, seja o movimento de um robô ou uma ação simulada na tela

do computador, parece que é o que prende o interesse do aluno." [56]

Outro motivo que tornou o uso do Robocode um sucesso foi a promoção de um ambiente competitivo em sala de aula, onde os alunos disputavam para ver quem conseguia escrever a melhor lógica. Porém, alguns problemas surgiram ao longo do percurso que inviabilizaram a utilização do jogo em semestres posteriores.

O Robocode é uma aplicação java relativamente simples, apresenta interface 2D e não possui som. Em [9] os autores afirmam que essa geração de alunos foi acostumada com jogos tridimensionais extremamente realísticos e talvez isso tenha diminuído um pouco o entusiasmo pelo experimento.

Outro problema foi que com o tempo, se descobriu que existia uma estratégia vencedora que era programar o tanque para ficar sempre próximo da parede da arena. Isso fez com que os alunos não tivessem incentivo para buscar diferentes formas para se resolver o problema.

Por último, como Robocode é um aplicativo relativamente conhecido, existe muito código publicado na internet e os professores não tinham certeza se trechos do código que estavam corrigindo eram ou não cópias.

A solução encontrada para resolver esses problemas foi implementar o Robocode no ambiente Muppets como podemos ver na figura 2.4. O Muppets é uma aplicação tridimensional capaz de produzir efeitos gráficos avançados, o que o torna interessante para uma audiência cada vez mais exigente devido ao apelo visual dos jogos.

A cada semestre também é possível desenvolver alterações em regras e características do jogo para que não surjam estratégias vencedoras, de forma a incentivar os alunos a procurem outras formas de abordar o problema. Seria possível por exemplo, criar um raio limitado de visão para o tanque, ou possibilitar a troca de armas durante o combate, ou até mesmo criar obstáculos no meio da arena. Adicionalmente, isso também resolve o problema da cópia de códigos.

2.4 Alice

Alice é uma ferramenta de prototipação rápida de animações tridimensionais voltada para leigos e iniciantes. Seus recursos de programação são intuitivos o bastante para que Alice seja usada como uma ferramenta de apoio em cursos de introdução à programação.

Figura 2.4: A arena do jogo Robocode implementado no Muppets. Vários tanques de guerra batalham entre sí. O tanque de guerra que prevalecer até o final é o vencedor. Cada aluno é responsável por escrever a lógica de seu tanque para competir com os colegas. [57].

2.4.1 Objetivo

O projeto Alice começou em meados de 1991 na universidade de Virginia, nos Estados Unidos [17]. A idéia de criar essa ferramenta surgiu quando os pesquisadores perceberam a dificuldade de se criar conteúdo para interfaces 3D interativas. Pessoas de 19 anos que não fossem programadores de computador e não fossem inclinados à matemática, não tinham acesso à ferramentas que possibilitassem a expressão de idéias por esse novo meio de comunicacão. Assim surgiu a motivação para criar o Alice.

O objetivo do Alice, é tornar intuitiva a programação de comportamentos de objetos em uma cena

tridimensional. Nas palavras de Conway "*Alice foca exclusivamente no problema de especificação de comportamentos por meio de controle programático*" [17].

2.4.2 Funcionamento

Com o objetivo de criar uma ferramenta intuitiva para iniciantes, o projeto Alice foi concebido com base nas seguintes premissas [25]:

- Fazer com que o estado dos objetos sejam tão visíveis ao aluno quanto for possível.
- Animar todas as mudanças de estado dos objetos.
- Não permitir que programas sintaticamente errados sejam escritos.
- Realçar a noção de Objeto (focando na programação orientada a objetos).
- Usar ambientes tridimensionais para motivar os alunos.

Na figura 2.5, podemos ver a interface do Alice em ação. A interface foi pensada para que o aluno não precise digitar os comandos, evitando que erros de sintaxe sejam cometidos. O aluno pode arrastar os comandos para dentro da caixa de script e a interface vai automaticamente abrir caixas de diálogo, caso algum parâmetro adicional seja necessário para o comando. Isto cria segurança no aluno porque ele não tem mais que lidar com as indecifráveis mensagens de erro apresentadas pelo compilador enquanto está tentando aprender os novos conceitos.

Essencialmente, o aluno está constantemente explorando possibilidades de alterar o comportamento e o estado de um objeto. O aluno pode escolher trabalhar com vários dos diversos objetos existentes na biblioteca do Alice e cada objeto vem com um conjunto de operações pré-estabelecidas. O aluno é encorajado a modificar essas características, criando oportunidades para o ensino de conceitos de orientação a objeto como herança e encapsulamento.

Na mesma linha que o projeto Muppets e o Projeto Soar/Games, os pesquisadores do projeto Alice citam diversas vezes as vantagens obtidas por meio da visualização dos programas [25, 19]. A possibilidade de fazer alterações no código e instantaneamente ver essas mudanças em atuação contribui para reter o interesse do aluno no problema em questão, inclusive fazendo com que os alunos dediquem tempo extra aos problemas apresentados em sala de aula [19].

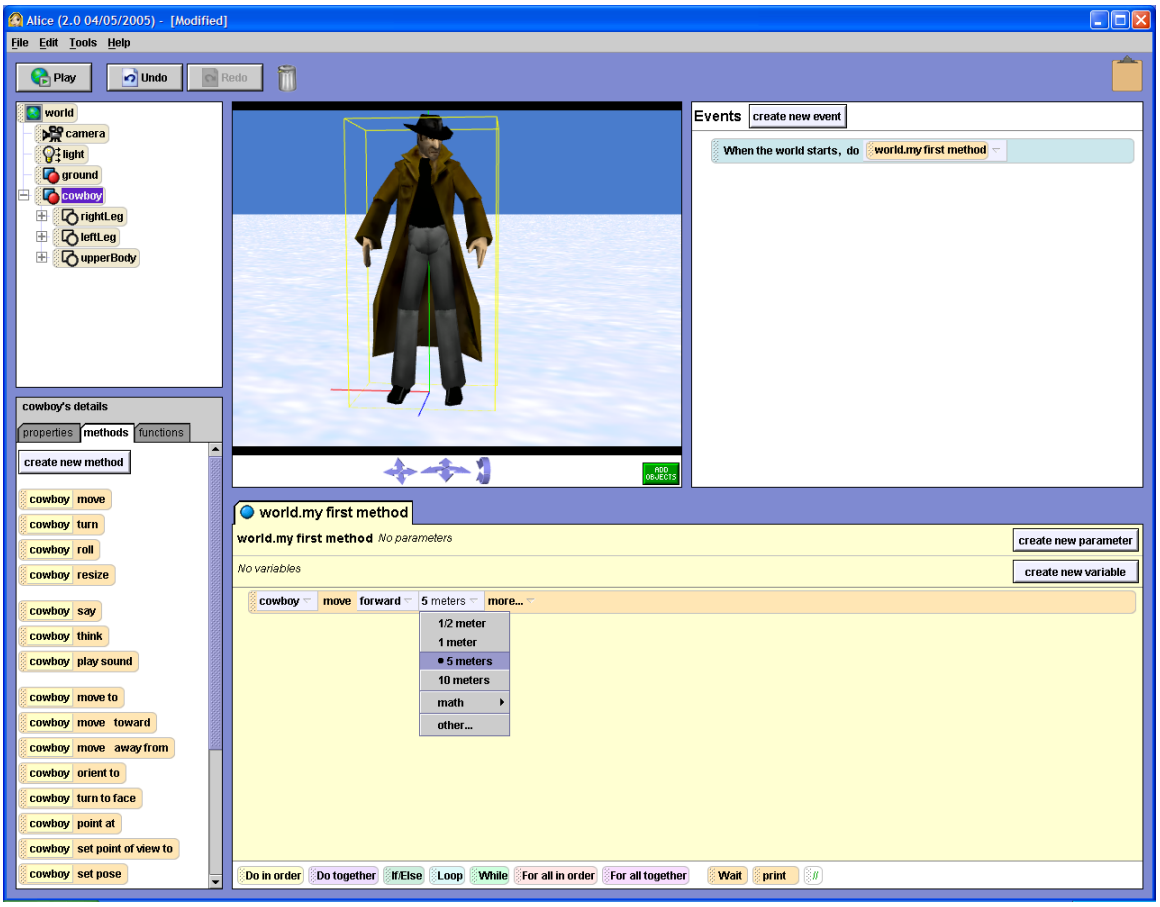

Figura 2.5: Interface do Alice. Os comandos à esquerda são arrastados para dentro da caixa de script, assim o aluno não precisa digita-los. Um botão play no canto superior esquerdo ativa a animação e o aluno pode ver os resultados de seu código.

Uma visão mais detalhada sobre o funcionamento do Alice e seus diversos recursos pode ser encontrada em [20], [23] e [24].

2.4.3 Resultados

Barbara Moskal, Deborah Lurie e Stephen Cooper conduziram um experimento em [51], que visava descobrir se conceitos fundamentais de programação poderiam ser ensinados para alunos sem experiência prévia com programação por meio do uso de uma nova metodologia de ensino que usa o Alice como ferramenta central.

Como resultado dessa pesquisa, foi descoberto que o método empregado aumentou a performance dos alunos, aumentou a retenção do curso e melhorou a atitude e a confiança dos alunos com relação à capacidade de terminar o curso de computação.

Diversas instituições estão utilizando o Alice para ministrar cursos introdutórios de computação e uma nova versão está sendo desenvolvida com o apoio da empresa norte americana *Eletronic Arts*. A nova versão vai apresentar uma integração com o jogo *The Sims*.

2.5 Considerações Finais

Como vimos, trabalhos importantes vêm sendo realizados na direção da utilização de jogos de computador e ferramentas interativas tridimensionais no ensino de Ciência da Computação e pesquisa aplicada em IA. Além dos trabalhos acima, temos outros exemplos que podem ser citados.

Tarski's World [8], é um aplicativo que busca usar a visualização para ensinar lógica de primeira ordem. Em *Tarki's World*, um aluno pode usar a linguagem disponibilizada pela ferramenta para descrever mundos populados por blocos de diferentes tamanhos e formatos. Esses blocos ficam em uma janela própria e podem ser visualizados em duas ou três dimensões, e podem ser construídos e modificados pelo aluno. Em uma outra janela, o aluno pode escrever e validar sentenças lógicas que fazem referência aos blocos exibidos na janela de blocos. A escrita das sentenças é facilitada por meio de um teclado visual que contém os conectivos lógicos. O desafio do aluno, é dizer se uma dada sentença é verdadeira ou não. Em caso de erro, a ferramenta mostra visualmente um contra exemplo que torna a sentença falsa.

Randolph Jones apresenta um curso de Projeto e Implementação de Jogos de Computador em [36], defendendo que o curso provê um ambiente ideal para os estudantes integrarem uma vasta base de conhecimentos e habilidades no curso de Ciência da Computação apresentando a variedade de conceitos de Ciência da Computação necessários para o desenvolvimento de um jogo.

O trabalho de Ron Coleman et al. [16], descreve o desenvolvimento de um currículo com concentração de disciplinas voltadas para o desenvolvimento de jogos de computadores.

Lasse Natvig e Steinar Line descrevem o AoC [52], *Age of Computers*, um jogo baseado em interface web onde podem ser ensinados conceitos fundamentais de computadores para um curso com 250 alunos. Os autores relatam que a resposta dos alunos foi bastante positiva e foi uma motivação forte para continuar com o projeto.

Dentro do ensino de computação, o foco deste trabalho está em utilizar jogos de computadores como ferramentas no ensino de Inteligência Artificial. A idéia central está na disponibilização de um ambiente virtual onde os alunos possam efetivamente construir e visualizar os agentes inteligentes em ação. Em particular, existe um gênero de jogo chamado *jogo de programação*, onde os jogadores programam a IA de seus agentes e os colocam para competir em um ambiente virtual. Podemos citar como exemplo *GUN-TACTYX* e o *Robocode*.

Capítulo 3

Tecnologias Utilizadas e Cenários de Aplicação

No capítulo anterior, vimos exemplos de como aplicar jogos ao ensino de computação. O tema comum que podemos extrair desses projetos, é que a visualização dos programas é o que parece capturar a atenção da pessoa que está observando ou interagindo com o sistema, como expressado em [56].

Nossa proposta tem o tema da visualização como aspecto central, e nesse capítulo, vamos mostrar algumas decisões de design que tivemos que fazer ao tentar formular uma arquitetura para uma ferramenta extensível que permita visualização tridimensional de programas PROLOG em cenários específicos. Também vamos apresentar as tecnologias com as quais vamos compor nossa aplicação. Começamos falando rapidamente sobre as principais idéias da linguagem PROLOG, bem como ver um pouco da sintaxe.

3.1 A linguagem PROLOG

PROLOG é um acrônimo que significa *PROgramming in LOGic* [65], ou programando em lógica. A idéia da programação lógica surgiu por volta de 1958, com um artigo do professor John McCarthy entitulado *Programs With Common Sense* [48]. Nesse artigo, o professor McCarthy introduz o programa *advice taker*, que influenciou posteriores desenvolvimentos na área de programação lógica.

A programação lógica faz sentido na medida em que traz elementos da lógica matemática para a programação de computadores. Parte do apelo à programação lógica vem do fato de que muitos problemas são expressos de maneira natural como teorias lógicas. É um estilo mais descritivo de programação, onde o programador precisa saber quais relações existem entre diversas entidades, enquanto na programação procedural, o programador precisa dizer exatamente o que o computador

precisa fazer, passo a passo.

A linguagem PROLOG viria a ser criada por Alain Colmerauer e Philippe Roussel em meados de 1972 usando o resultado de Robert Kowalski sobre a interpretação procedural das Cláusulas de Horn [40]. Ela não é a única linguagem de programação lógica, mas muitas dessas linguagens são baseadas em PROLOG [13].

Cabe aqui uma explicação sobre o vínculo das Cláusulas de Horn com o conceito de programação lógica, e com a linguagem PROLOG. Para programar em lógica, deve existir uma maneira de testar se uma dada formula lógica é verdadeira ou falsa.

Ou seja, queremos achar um conjunto de valorações para os termos de uma fórmula lógica de tal forma que o valor da fórmula seja verdadeiro. Esse é um problema particularmente difícil. O problema continua difícil se fecharmos o escopo em fórmulas booleanas, onde cada variável pode ser verdadeira ou falsa.

O problema de decisão de responder se existe ou não uma valoração que torne uma fórmula booleana verdadeira tem o nome de problema da satisfazibilidade booleana, um problema muito importante na área de Algoritmos e Estudos de Complexidade. O cientista Stephen Arthur Cook provou [18] que esse problema é NP-Completo[1], e portanto considerado intratável.

Esse problema de decisão tem um caso particular chamado de problema da k-satisfazibilidade. No problema da k-satisfazibilidade (k-SAT), trabalhamos com formulas proposicionais na forma normal conjuntiva, que é uma conjunção de cláusulas de literais. A fórmula tem o seguinte formato:

$$(\chi_{1,1} \vee \chi_{1,2} \vee \ldots \vee \chi_{1,k}) \wedge (\chi_{2,1} \vee \chi_{2,2} \vee \ldots \vee \chi_{2,k}) \wedge \ldots \wedge (\chi_{n,1} \vee \chi_{n,2} \vee \ldots \vee \chi_{n,k}) \tag{3.1}$$

Na formula 3.1, podemos substituir quaisquer $\chi_{i,j}$ por $\neg\chi_{i,j}$, sem sair da forma normal conjuntiva. Chamamos $\chi_{i,j}$ de literal. Para $k = 3$, o problema continua sendo NP-Completo, como demonstrado por Richard Manning Karp em [38]. Em contraponto, se dissermos que toda cláusula da fórmula proposicional é uma Cláusula de Horn, isto é, uma cláusula que tem no máximo um literal positivo, então o problema de satisfazer essa fórmula, chamado de HORNSAT, é um problema P-completo. Portanto, podemos afirmar que existe uma solução para HORNSAT em tempo polinomial. Esse é

[1]Nesse mesmo artigo[18], a definição de NP-Completude foi formalizada.

um fato importante porque permite que as técnicas de resolução para esse tipo de fórmula sejam executadas em um limite de tempo razoável. As cláusulas de Horn têm esse nome por causa do cientista Alfred Horn, que foi o primeiro a apontar a significância desse tipo de cláusulas em 1951 [31].

A relação entre PROLOG e as cláusulas de Horn reside no fato de que um programa PROLOG é uma conjunção de Cláusulas de Horn. Uma Cláusula de Horn,

$$\neg\alpha \vee \neg\beta \vee \ldots \vee \neg\gamma \vee \chi \tag{3.2}$$

pode ser reescrita da seguinte maneira:

$$\chi \longleftarrow (\alpha \wedge \beta \wedge \ldots \wedge \gamma) \tag{3.3}$$

Isso significa que para provar[2] χ deve-se provar α e β e ... e γ. Em PROLOG a sintaxe da fórmula 3.3 fica da seguinte maneira, observando que toda sentença em PROLOG termina com um ponto:

```
x :- a, b, ..., z.
```

Essa construção em PROLOG é chamada de *regra*. Além disso, o PROLOG dispõe de outras caraterísticas importantes. O PROLOG é uma linguagem baseada na lógica de predicados de primeira ordem [13]. Um predicado é a expressão da relação entre entidades. Por exemplo, você pode dizer que a cor de um carro é vermelha da seguinte maneira:

```
cor(carro, vermelha).
```

Desse modo, o predicado `cor` exprime uma relação entre as constantes `carro` e `vermelha`. Essa construção, que não possui implicação, é chamada em PROLOG de *fato*. Podemos dizer que `carro` e `vermelha` são os termos do predicado. Podemos também dizer que o predicado `cor` tem *aridade* 2. A aridade de um predicado é um número, e indica quantos termos o predicado possui. Podemos dizer isso de outra forma: `cor/2` indica um predicado `cor` com 2 termos.

[2] ou *mostrar*

Em PROLOG temos também o conceito de variáveis. Suponha que se queira saber qual é a cor do carro[3]. Podemos escrever a *consulta*:

```
1 ?- cor(carro, X).
```

Em PROLOG, as variáveis começam com letra maiúscula. Ora, dados os dois predicados acima, é fácil entender que $X = vermelha$. O PROLOG faz isso por meio de um mecanismo chamado de *unificação*, um problema de computação muito importante. O objetivo da unificação é *casar* termos por meio de substituição das variáveis por valores específicos. O nome *unificação* foi usado pela primeira vez pelo cientista J. Alan Robinson em [61]. Dizemos que para os predicados $f(X, a) = f(a, X)$, temos uma *substituição* $\{X \longmapsto a\}$ que *unifica* os termos dos predicados, com o resultado $f(a, a) = f(a, a)$. Um exemplo pode ser:

```
cor(carro,vermelha).
1 ?- cor(carro,X).
X = vermelha;
```

Podemos ter casos onde o PROLOG não consegue unificar ou provar uma sentença. Por exemplo:

```
cor(carro,vermelha).
1 ?- cor(carro,vermelha).
Yes
2 ?- cor(carro,azul).
No
3 ?-
```

Aqui podemos ver que na primeira consulta, o PROLOG consegue unificar a sentença facilmente, uma vez que os predicados são exatamente iguais. Já na segunda consulta, é impossível unificar os predicados, e portanto o PROLOG gera uma falha. É importante saber que o PROLOG tenta novamente unificar a sentença por meio de um mecanismo de *backtracking*. Por exemplo no trecho abaixo:

[3] A maioria das implementações apresentam um modo interativo onde você pode digitar esse tipo de *consulta*.

```
cor(verde).
cor(azul).
cor(vermelha).
1 ?- cor(vermelha).
Yes
2 ?-
```

Podemos ver que o PROLOG consegue realizar a unificação. Ele falha na tentativa $cor(verde) = cor(vermelha)$, falha na tentativa $cor(azul) = cor(vermelha)$ e consegue unificar com sucesso $cor(vermelha) = cor(vermelha)$.

Um predicado especial chamado de *corte*, representado pelo símbolo `!`, pode ser usado para controlar o mecanismo de *backtracking* do PROLOG. Para exemplificar como o corte funciona, suponha o seguinte exemplo extraído de [13]:

```
imposto(X,Y):-
  brasileiro(X),
  devedor(X,Y).
imposto(X,Y):-
  sueco(X),
  devedor(X,Y).
```

Nesse exemplo, se `X` for brasileiro mas não for devedor, o PROLOG vai desperdiçar um teste para ver se o `X` é sueco. Com a adição do corte no lugar correto podemos controlar o comportamento do backtracking de forma que o PROLOG não gaste processamento com esse teste, como no exemplo:

```
imposto(X,Y):-
  brasileiro(X),
  !,
  devedor(X,Y).
imposto(X,Y):-
  sueco(X),
  !,
  devedor(X,Y).
```

Aqui estamos dizendo para o PROLOG aceitar todas as escolhas que foram feitas até o momento para a cláusula atual e para não considerar outras cláusulas no caso dessa falhar. Então se `X` for brasileiro e não for devedor, `imposto/2` falha imediatamente, sem tentar unificar a próxima cláusula que testaria `sueco(X)`, economizando processamento.

Com essa base, podemos introduzir a forma pela qual o PROLOG implementa a negação. A negação é implementada em PROLOG por meio de falha. A idéia é que um predicado de negação falhe ao tentar algo verdadeiro e tenha sucesso ao tentar provar algo falso. Fazemos isso com o uso do corte e com auxílio do predicado `fail/0`. O predicado `fail/0` falha sempre, e é muito útil como veremos no exemplo:

```
negar(X) :-
  X,
  !,
  fail.
negar(X).

cor(azul).
cor(preto).

1 ?- negar(cor(verde)).
  Yes
2 ?- negar(cor(azul)).
  No
3 ?-
```

Podemos ver acima, que o predicado `negar/1` teve sucesso quando fizemos a consulta: "*é falsa a afirmação que verde é uma cor?*" representada pelo predicado `negar(cor(verde))`. Também podemos conferir o passo contrário: `negar(cor(azul))` falha porque no exemplo, azul é de fato uma cor.

O padrão de chamada dos argumentos de um predicado, onde se verifica qual argumento é de entrada e qual é de saída, é um conceito importante em PROLOG, principalmente para pessoas com experiência em linguagens de paradigma estruturado. As funções, no paradigma estru-

turado, têm uma sintaxe específica para o retorno, como por exemplo `int func();` em C [39]. Também podemos retornar resultados por meio da passagem de parâmetro por referência, como em `void func(SomeType &v);` ou ainda `void func(SomeType *v);`. A passagem de parâmetros e retornos em PROLOG é realizada nos argumentos do predicado.

Por exemplo, podemos ter um predicado como `sucessores/3`, onde dado um número `X`, ele nos informa quais são seus dois próximos sucessores `Y` e `Z`:

```
00  sucessores(X,Y,Z):-
01   Y is X + 1,
02   Z is X + 2.
03  1 ?- sucessores(1,A,B).
04    A = 2
05    B = 3
06  2 ?- sucessores(1,2,3).
07    yes
08  3 ?- sucessores(A,2,3).
09    ERROR: is/2: Arguments are not sufficiently instantiated
```

Nesse caso, o `X` é o argumento de entrada, enquanto `Y` e `Z` são unificados com o resultado, atuando assim como os *retornos* do predicado, fazendo um paralelo com as linguagens estruturadas. Podemos introduzir o conceito de instanciação de um argumento. Na consulta 1 acima, dizemos que o primeiro argumento do predicado `sucessores/3` está *instanciado* com o número 1, enquanto os outros dois não estão instanciados. O objetivo do PROLOG nesse caso, será unificar os dois últimos argumentos, com algum valor.

Podemos usar uma notação para expressar essa relação, `mode sucessores(+,?,?)`.O símbolo `+` indica um argumento de entrada, que deve ser instanciado. O símbolo `-`, indica um argumento que será unificado com o resultado. O símbolo `?` significa que o argumento pode ser ou não instanciado. A notação `mode`, é usada para documentar os predicados em um arquivo de código por meio de comentários [13].

No código acima vemos que na consulta 2, todos os termos estão instanciados. Estamos verificando se os números 2 e 3 são sucessores de 1. Na consulta 3, estamos usando o predicado de maneira incorreta, porque o primeiro argumento é a variável `A`, e portanto não é instanciado. O erro acontece

logo na linha 01, porque o predicado especial usado nas linhas 01 e 02, `is/2`, é `mode is(?,+)`. Portanto, ele exige que o segundo termo seja instanciado, o que não vai acontecer nesse caso. Deste modo, ocorre um erro conforme mostra o prompt do SWI-Prolog na linha 09.

O PROLOG disponibiliza predicados para alterar regras e fatos no programa: `assert/1` que tem `mode assert(+)`, e `retract/1` que tem `mode retract(?)`. Isso é útil porque podemos usar esses predicados como formas de variáveis globais ou *flags* [13]. Vamos ver o seguinte exemplo:

```
00 1 ?- cor(azul).
01   ERROR: Undefined procedure: cor/1
02 2 ?- assert(cor(azul)).
03   Yes
04 3 ?- cor(azul).
05   Yes
06 4 ?- retract(cor(X)).
07   X = azul
08 5 ?- cor(azul).
09   No
10 6 ?-
```

No exemplo acima, podemos observar o efeito dos predicados `assert/1` e `retract/1`. Na linha 00, tentamos verificar se azul é uma cor. Como esse *fato* não existe ainda no programa, o prompt acusa um erro na linha 01. Na linha 02, usamos o predicado `assert/1` para definir dinamicamente o predicado `cor(azul)` enquanto estamos no *prompt*. Na linha 04, testamos novamente e desta vez o PROLOG consegue unificar `cor(azul)`. Na linha 06, usamos o predicado `retract/1`, com o objetivo de apagar dinamicamente, todos os *fatos* `cor/1`. Quando testamos novamente, na linha 08, o PROLOG apenas não consegue unificar o fato, diferentemente do que aconteceu na linha 01. Isso acontece porque o predicado `retract/1` apaga as instâncias de `cor/1`, mas não a definição.

A linguagem PROLOG também manipula predicados e termos extra-lógicos que têm o objetivo de efetuar operações computacionais convencionais existentes em praticamente todas as linguagens de programação. Por exemplo, temos predicados que enviam uma mensagem para um dispositivo de saída. O exemplo a seguir imprime *Olá* no console do usuário:

```
format("Olá!",[]).
```

Existe uma extensa bibliografia cobrindo a linguagem PROLOG. Para um maior aprofundamento na linguagem, indicamos um texto online introdutório do Professor Paul Brna, *PROLOG Programming: a first course*, que se encontra disponível em [13]. Indicamos também, *The Art of Prolog*, um livro que apresenta um bom nível de detalhe tanto em assuntos práticos quanto teoricos, de Ehud Shapiro e Leon Sterling [63].

3.2 Os Cenários de Aplicação

Como dito anteriormente, o objetivo do trabalho é construir um ambiente tridimensional com cenários específicos habitados por agentes. Os agentes têm objetivos bem definidos nesses cenários. Queremos fazer com que a inteligência da qual os agentes dispõem para alcançar os objetivos do cenário esteja codificada em um programa PROLOG. Vamos agora discutir os dois cenários que a ferramenta vai implementar. Esses cenários foram escolhidos em virtude de cobrirem uma boa parte dos temas de um curso de lógica para graduação. Apresentamos *O problema do Labirinto* e *O Mundo de Wumpus*.

3.2.1 O Labirinto

O cenário do Labirinto contém um problema clássico da computação na área de grafos que é achar o menor caminho entre dois vértices em um grafo. Especificamente, o cenário é um plano quadriculado $n \times n$ em um espaço tridimensional onde cada quadrado seria um vértice do grafo e a ligação entre dois quadrados seriam as arestas. Um agente habita esse espaço e seu objetivo é andar do quadrado que está para um outro qualquer, escolhendo o menor caminho possível entre os dois quadrados e desviando-se de quaisquer obstáculos existentes no percurso.

No começo da simulação, o agente estará parado em algum quadrado arbitrário no quadriculado. Além disso, inicialmente o quadriculado não possuirá obstáculos. O agente possui visão de toda a superfície do plano incluindo quaisquer eventuais obstáculos.

O aluno tem a opção de construir obstáculos nesse plano clicando arbitrariamente nos quadrados de sua escolha de modo a dificultar o trabalho do agente de ir do quadrado que ele está ao de destino. O aluno poderá também a qualquer momento, mandar o agente mover-se para outro quadrado que não esteja obstruído por um obstáculo. O agente não se move sozinho, ele apenas obedece essas ordens do aluno.

Como mostra a figura 3.1, podem existir vários caminhos possíveis para ir do ponto A para o

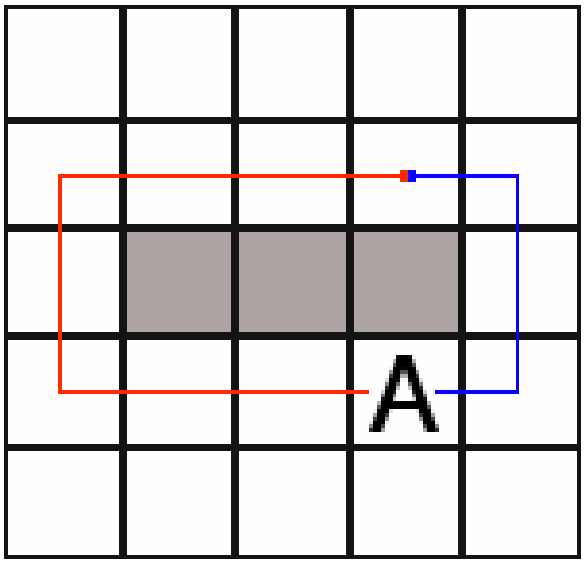

Figura 3.1: O plano quadriculado. Aqui mostramos dois possíveis caminhos até o destino. Os quadrados em cinza são os obstáculos e o Agente está representado pela letra *A*. O caminho vermelho leva o agente até o destino apesar de o caminho azul ser uma escolha melhor.

ponto *B*. Pode ser também que um caminho seja melhor que os outros. Vamos optar por possibilitar 8 direções a se seguir a partir de um quadrado: 2 na vertical, 2 na horizontal e 4 nas diagonais.

A lógica de como o agente fará para encontrar o melhor caminho entre sua posição e o destino será escrita pelo aluno em linguagem PROLOG. Uma vez escrita a lógica em PROLOG, o aluno poderá usar a ferramenta para ver se sua lógica está funcionando de acordo. Tudo que o algoritmo precisa fazer é encontrar uma lista de quadrados que percorridos em seqüência, levam o agente ao destino.

3.2.2 O Mundo de Wumpus

O jogo *Mundo de Wumpus*, cujo título original em inglês é *Hunt the Wumpus*, foi criado pelo designer de jogos Gregory Yob e publicado em 1972 [4]. Uma descrição do jogo também pode ser encontrada no livro de Russel e Norvig [62].

Wumpus é um monstro atroz que vive em uma caverna escura cheia de precipícios mortais, espreitando aventureiros que ousem adentrá-la. O Wumpus exala um cheiro muito forte e característico. Os precipícios são profundos o suficiente para que se possa perceber uma leve brisa em suas proxi-

midades. Uma pepita de ouro reluzente se encontra em algum lugar na caverna. O aventureiro pode desferir um potente golpe contra o monstro usando suas próprias mãos. Como o golpe é potente e consome muita energia, o aventureiro poderá usá-lo uma única vez.

O objetivo do aventureiro é entrar na caverna, conseguir recuperar a pepita de ouro, e voltar para a saída. O aventureiro deve evitar estar no mesmo quadrado que o monstro, porque nesse caso ele sucumbirá diante da criatura. Se o aventureiro cair no precipício, ele encontrará seu fim.

Esse cenário também pode ser implementado em um quadriculado $n \times n$. Se o agente estiver no mesmo quadrado que a pepita de ouro, ele pode pegá-la. O aventureiro consegue sentir uma leve brisa nos quadrados adjacentes aos precipícios. O aventureiro também consegue sentir o forte odor exalado pelo Wumpus caso esteja em um quadrado adjacente.

O golpe desferido pelo aventureiro subjugará o monstro caso o aventureiro esteja em um quadrado adjacente e posicionado de frente para a criatura. Ocorrendo a morte do Wumpus, seu odor característico se dissipa rapidamente nos quadrados adjacentes. O aventureiro só conhece esses fatos, e sua visibilidade está restrita ao quadrado que ele está ocupando no momento.

A figura 3.2 mostra uma possível configuração do cenário do mundo de Wumpus.

O que descrevemos acima, foi uma versão levemente modificada do Mundo de Wumpus para atender as necessidades de modelagem do presente trabalho[4]. A principal modificação foi que em vez de se valer de um arco, o aventureiro é dono de um potente soco que só eliminará o Wumpus caso seja desferido de uma posição adjacente. Na versão original, a flecha poderia ser disparada de qualquer local da caverna.

3.3 *Engines* Gráficas

Programar uma aplicação tridimensional é uma tarefa complexa e computacionalmente custosa. Tanto que por natureza, as aplicações tridimensionais são altamente beneficiadas quando se valem de aceleração tridimensional em *hardware*, por serem intensas consumidoras de recursos computacionais em virtude dos cálculos geométricos envolvidos.

Com isso, foi natural que a indústria criasse APIs para a programação 3D com o objetivo de abstrair os detalhes de *hardware* de cada fabricante. As Principais APIs existentes hoje no mercado

[4]O autor não é versado na arte da modelagem tridimensional, embora tentativas tenham sido feitas nesse sentido. Ademais, encontrar um modelo tridimensional razoável e gratuito que possua a animação de um disparo de arco se mostrou uma tarefa bastante desafiadora.

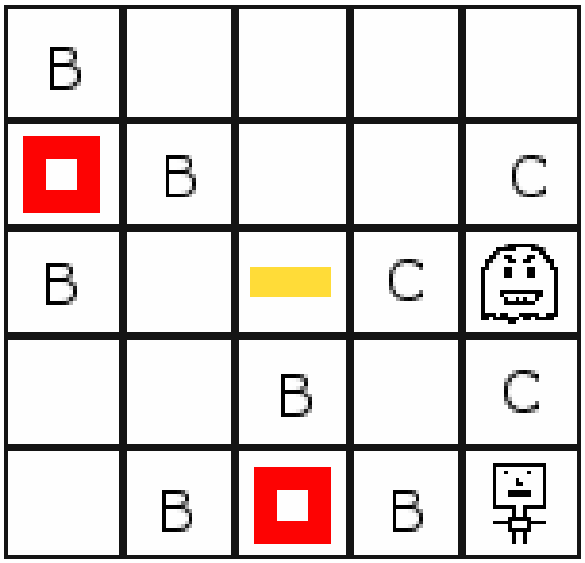

Figura 3.2: Um possível cenário no mundo de Wumpus. O agente pode ser visto no canto inferior direito. O Wumpus se econtra logo dois quadrados acima. A letra *C* representa o odor exalado pelo Wumpus. Os quadrados com um furo no centro representam os precipícios e as letras *B* em suas adjacências, representam a brisa soprando. Por fim, o pequeno retângulo no centro do quadriculado representa a pepita de ouro.

são a OpenGL criada pela Silicon Graphics(SGI) e a Direct3D da Microsoft. Essas APIs constituem o nível mínimo de abstração normalmente adotado quando se deseja construir programas que manipulam primitivas tridimensionais.

Esse nível de abstração pode ser muito baixo dependendo da aplicação que se quer construir. No caso particular da ferramenta proposta, queremos que o modelo tridimensional do agente seja por exemplo um robô no caso do cenário do labirinto, e um aventureiro humano no caso do mundo de Wumpus. O Wumpus mesmo poderia ter um modelo de um monstro como um *goblin*[5]. Uma vez que as primitivas disponíveis pelas APIs são muito simples para construir modelos mais refinados como esses, construir a ferramenta partindo dessas APIs seria uma tarefa de alta complexidade.

O mais indicado para construir esses modelos, é usar programas de modelagem tridimensional como por exemplo o 3D Studio MAX da AutoDesk, ou o Blender como uma opção de Software Livre. Depois de construído o modelo, pode-se exporta-lo para uma *3D Engine*.

[5]Uma Enciclopedia com informações sobre goblins, entre outras criaturas e temas correlatos, pode ser encontrada em [33]

As *3D Engines* ou *Graphic Engines* estão no próximo nível de abstração, e são aplicativos que possuem diversas funcionalidades que aceleram o ciclo de desenvolvimento de uma aplicação tridimensional. Uma delas em particular, é a possibilidade do carregamento do modelo em um formato que a *engine* reconhece. Para esse fim, é comum que uma *engine* madura disponibilize um *plugin* para os principais softwares de modelagem tridimensional do mercado. Assim pode-se trabalhar no software de modelagem e quando o modelo tridimensional estiver pronto, pode-se exporta-lo para o formato reconhecido pela *engine* por meio do *plugin*.

Uma *Game Engine* contém funcionalidades de uma *Graphic Engine* e implementa mais funcionalidades específicas de jogos como por exemplo som, algoritmos de IA, interface com o usuário entre outros. Existem centenas de *engines* disponíveis com diversos modelos de licensa e faixas de preço. Na parte de software livre, podemos citar *OGRE* e *Panda3D*. A *Torque Game Engine* da empresa *GarageGames* é uma *engine* comercial bastante popular. *Unreal Engine* da empresa *Epic Games* e a CryENGINE2 desenvolvida pela Crytek, são exemplos de *Game Engines* comerciais de última geração.

A OGRE em particular é uma *engine* gráfica popular distribuída sob licensa LGPL, e é madura o suficiente para as necessidades deste trabalho. A *engine* oferece suporte para os principais softwares de modelagem do mercado e já foi utilizada com sucesso em diferentes projetos comerciais além de possuir uma comunidade ativa. A OGRE trabalha com a linguagem C++[6] e será a *engine* escolhida para construir a ferramenta.

Descartamos desse modo a possibilidade de construir a ferramenta em PROLOG, porque apesar de existirem *bindings* para alguns dialetos e extensões da linguagem, em particular a implementação do professor Gildas Ménier [49] para suportar OpenGL no Visual PROLOG, vimos que existem opções mais interessantes nesse contexto. Também é importante frisar que não temos conhecimento da existência de uma *engine* gráfica madura para PROLOG. Logo, como sabemos que vamos ter uma aplicação externa ao PROLOG feita em C++, precisamos econtrar uma maneira de integrar o PROLOG com C++.

[6]Existem projetos para construir *bindings* da OGRE para outras linguagens como Java. Entretanto, esses projetos se encontram ainda incipientes em 2007.

3.4 Integrando SWI-Prolog e C++

Na seção anterior vimos o porquê da necessidade de integrar uma aplicação C++ com PROLOG. Nessa seção veremos os detalhes de implementação do SWI-PROLOG que permitem que essa integração seja realizada.

Existem várias implementações de PROLOG disponíveis. No grupo de implementações comerciais podemos citar o Quintus PROLOG e o SICStus PROLOG, desenvolvidos pelo Instituto Sueco de Ciência da Computação. O SWI-Prolog, desenvolvido na Universidade de Amsterdam, e o *Ciao Prolog Development System* desenvolvido na Universidade Politécnica de Madrid, são duas implementações de PROLOG disponíveis sob licensa LGPL.

O SWI-Prolog, cujo principal autor é o professor Jan Wielemaker, vem sendo continuamente desenvolvido desde 1987. SWI é um acrônimo em holandês que significa *Sociaal-Wetenschappelijke Informatica*[7] ou Informática de Ciências Sociais, e é o nome de uma divisão de pesquisa da Universidade de Amsterdam. O SWI-Prolog é amplamente usado tanto na pesquisa e ensino quanto em aplicações comerciais e é um implementação de software livre bastante popular [69]. O SWI-Prolog será nossa implementação escolhida para o desenvolvimento da ferramenta.

Todas as implementações de PROLOG citadas disponibilizam alguma forma de integração com outras linguagens, e a maioria das outras implementações atuais que não citamos também têm essa funcionalidade. Não existe um padrão definindo como deve ser feita essa integração, então no geral, cada implementação de PROLOG faz isso de uma maneira diferente.

O SWI-Prolog em particular, faz essa integração por meio de seu mecanismo chamado de *Foreign Language Interface*, ou Interface para Linguagens Externas. Detalhes podem ser econtrados no manual do SWI-Prolog em [2]. Essa interface permite que o SWI-Prolog faça chamadas ao C e vice-versa. Na implementação, vamos precisar fazer com que o C faça chamadas ao PROLOG.

O motivo de precisarmos fazer com que o C faça chamadas à predicados PROLOG reside no seguinte ponto de vista arquitetural. Na aplicação C++, teremos toda a lógica dos cenários, incluindo o conhecimento de quando as animações iniciam e terminam em função da ação do usuário. No caso do labirinto por exemplo, quando o usuário clicar em algum quadrado arbitrário, queremos que o agente descubra qual é o menor caminho para chegar até lá partindo de onde ele se encontra. Faz sentido que, no momento do clique, a aplicação C++ busque unificar um predicado PROLOG de

[7] Em 2007 o SWI se chama HCS, de *Human Computer Studies Laboratory*.

forma a obter uma lista de posições representando o caminho a ser percorrido pelo agente, por exemplo:

```
findPath(Origem, Destino, Path).
```

Onde `Path` é a lista de quadrados que leva da `Origem` para o `Destino`. Nessa linha, podemos dizer que a aplicação C++ vai usar SWI-Prolog como um componente auxiliar nos momentos em que precisar obter efetivamente decisões inteligentes do agente, que o levarão a completar seu objetivo. Nesse exemplo do labirinto, tudo que o usuário precisaria desenvolver seria o predicado `findPath/3` `mode findPath(+,+,-)`.

Para fazer com que o C++ use o PROLOG como uma aplicação auxiliar, ou seja, para *embarcar* a *engine* do SWI-Prolog em uma aplicação C++, deve-se fazer a seguinte chamada antes de qualquer outra chamada à interface do PROLOG:

```
#include<SWI-cpp.h>
int main() {
    PlEngine e("");
    //resto do código
}
```

Com esse código, o C++ tenta em algum momento buscar o kernel de runtime, representado `libpl.dll`, no diretório corrente, e aassim a *engine* é carregada junto com a aplicação C++ e fica pronta para eventuais consultas. Em uma instalação padrão do SWI, a `libpl.dll` fica no diretório bin. Nesse exemplo, a aplicação deve estar nesse diretório para que o mecanismo funcione. As chamadas à engine do SWI acontecem na mesma *thread* do game loop, então a ferramenta Odin fica esperando a execução da *engine* do SWI para continuar a simulação.

Para exemplificar como a aplicação C++ faria para chamar o predicado `findPath/3`, vejamos a seguinte função[8]:

```
#include<SWI-cpp.h>
```

[8]Na seção A.1 do apêndice A, mostramos o código completo da classe PrologAdapter, que usamos para fazer a integração com o SWI-Prolog

```
    bool unifyPredicate(const String& predicate, vector<String> &args) {
00    int b = 0;
01    int max = static_cast<int>(args.size());
02    {
03        PlFrame fr;
04        PlTermv pv(max);
05        for(int i = 0; i < max; i++)
06            if(args[i].size() > 0) {
07                if(isCompound(args[i])) {
08                    String s = args[i];
09                    pv[i] = PlCompound(s.c_str());
10                } else
11                    pv[i] = args[i].c_str();
12            }
13        PlQuery q = PlQuery(predicate.c_str(), pv);
14        b = q.next_solution();
15        for(int i = 0; i < max; i++)
16            args[i] = pv[i];
17    }
18    return b != 0;
    }

    bool isCompound(const String& pred) {
19    size_t k = pred.find('(');
20    if (k != pred.npos)
21        return true;
22    return false;
    }
```

Estamos usando nesse código, as classes disponibilizadas pelo SWI-Prolog para trabalharmos com C++. A interface do SWI-Prolog na verdade é composta por funções em C. As classes `PlFrame` e `PlTermv` que estamos usando na função `unifyPredicate`, fazem parte de um *Wrapper* dessas funções

para o uso em C++, escrito no *header* `SWI-cpp.h` que pode ser encontrado no diretório *include* na instalação padrão do SWI-Prolog.

Vejamos agora o que está acontecendo no código da função `unifyPredicate` acima. Na linha 03, declaramos uma instância de `PlFrame` dentro de um bloco aparentemente desnecessário, entre as linhas 02 e 17. `PlFrame` tem um efeito importante nas variáveis do tipo `PlTerm`, no caso representadas na linha 04 por `PlTermv`, que é um vetor de `PlTerm`. `PlTerm`, é uma referência temporária para um termo que existe em uma pilha de execução na instância do PROLOG. O escopo dos termos criados no C++ na linha 04 é definido pelo escopo da instância `fr` de `PlFrame` definida na linha 03. Ou seja, estamos dizendo que todos os termos criados da linha 04 à linha 16, sejam descartados na linha 17, que é precisamente quando o escopo da instância de `PlFrame` acaba.

Nas Linhas de 05 a 12, estamos preenchendo o vetor de termos `pv` com os argumentos do parâmetro `args`. A classe `PlTerm` exige que se o argumento for um termo composto, então devemos utilizar o construtor de `PlCompound` como observado na linha 09. Para descobrir se o termo é composto ou não, procuramos pelo carácter do parênteses, que é o que a função `isCompound` está fazendo na linha 07.

Depois de termos preenchido o vetor de termos `pv`, montamos uma consulta PROLOG por meio da classe `PlQuery`, passando como parâmetro esse vetor e também o predicado que queremos consultar, representado pela variável `predicate` do tipo `String`, como mostra a linha 13. O método `c_str()` de `String` retorna a representação da `String` em um *array* de ponteiros para o tipo `char` do C. Em seguida, na linha 14 o método `next_solution` de `PlQuery` é invocado. Esse método delega para o PROLOG a consulta que estamos querendo fazer. Se o número retornado for diferente de 0, é porque o PROLOG conseguiu unificar, caso contrário, significa que o PROLOG não conseguiu unificar.

Quando a função `next_solution` retorna e o PROLOG consegue unificar, o vetor de termos `PlTermv` contém a resposta. Nas linhas 15 e 16 estamos colocando essa resposta na variável `args`, parâmetro passado por referência para a função `unifyPredicate`. Finalmente na linha 18, retornamos um `boolean` em função do resultado da operação.

Vejamos agora um possível trecho de código que usa a função `unifyPredicate`, para consultar o predicado `findPath(+,+,-)`:

```
00  vector<String> sv;
01  sv.resize(3);
```

```
02  sv[0] = "coords(0,0)";
03  sv[1] = "coords(0,2)";
04  sv[2] = "";
05  unifyPredicate("findPath", sv);
06  String path = sv[3];
```

Uma interface deve ser definida tanto para os argumentos de entrada quanto nos argumentos de saída do predicado. Neste caso, os argumentos de entrada são dois pontos: a as coordenadas da posição inicial e da posição final. O C++ está usando o formato `coords(X,Y)`, então o código escrito em PROLOG deverá levar isso em consideração. O retorno também deve seguir essa regra. Por exemplo poderíamos neste caso retornar tanto uma lista de direções como `[d1,d2,...,dn]` como uma lista de coordenadas `[coord(0,1),coord(0,2)]`. As direções `d1,d2,...,d8` representariam as oito direções de liberdade de movimento em um plano quadriculado. Essa resposta vem para o C++ em formato de *string*, como mostra a linha 06 acima, sendo que um *parsing* deve ser feito para extrair as coordenadas.

Como o predicado `findPath/3` tem `mode findPath(+,+,-)`, os dois primeiros argumentos devem ser preenchidos, como mostra as linhas 02 e 03 do código acima. Na linha 04, preenchemos o terceiro argumento com uma *string* vazia. Isso significa para o PROLOG que esse termo é uma variável, e o PROLOG tentará unifica-lo com uma resposta. No nosso caso, a resposta será o caminho, que vem em forma de *String* como podemos ver na linha 06.

No cenário do labirinto, cenário onde o predicado `findPath/3` vai se fazer necessário, temos a possibilidade de ter barreiras. Como essas barreiras podem ser construídas e destruídas pelo usuário dinamicamente, devemos ter uma maneira de comunicar esse tipo de evento para o PROLOG partindo do C++. Fazemos isso empregando os predicados `assert/1` e `retract/1`. Quando o usuário clica em um quadrado indicando que ele quer uma barreira na posição, podemos usar o `assert/1` para adicionar esse *fato* ao PROLOG. Por exemplo, poderíamos fazer `assert(obstacle(0,0))`, indicando que a coordenada `(0,0)` é uma barreira. Vejamos como isso é feito em C++ com o código a seguir:

```
01  int assertSomething(const String& something) {
02    int i = 0;
03    {
04      PlFrame fr;
```

```
05      PlTermv t1;
06      if(isCompound(something))
07        t1 = PlCompound(something.c_str());
08      else
09        t1 = something.c_str();
10      PlQuery q("assert", t1);
11      i = q.get_next_solution();
12    }
13    return i;
14  }
```

Esse trecho de código é semelhante ao trecho da função `unifyPredicate` mostrada acima. A principal diferença é que não precisamos retornar um resultado. Esse código faz com que o fato que está na variável `something` da função `assertSomething` seja adicionado ao PROLOG. Assim, supondo que `something="obstacle(0,0)"`, depois de executada essa função, no lado do PROLOG uma consulta por `obstacle(X,Y)` unificaria com `X=0` e `Y=0`. Da mesma maneira, se trocarmos `assert` por `retract` na linha 10 poderíamos retirar um fato do PROLOG, por exemplo, quando o usuário decidisse que uma dada posição não seria mais uma barreira.

3.5 A *Engine* de Renderização Gráfica OGRE

A OGRE é uma *engine* de renderização Gráfica orientada a cenas[9], escrita em C++ e disponível sob licensa LGPL. O projeto foi iniciado por Steve Streeting em meados de 2000 e nasceu a partir da idéia de criar uma *engine* gráfica Orientada a Objeto com o objetivo de abstrair a complexidade da API Direct3D. Posteriormente, a OGRE evoluiu adicionando suporte para a API OpenGL. OGRE é um acrônimo que significa *Object-Oriented Graphics Rendering Engine*, ou *Engine* de Renderização Gráfica Orientada a Objetos.

O objetivo da OGRE é facilitar o trabalho dos desenvolvedores de aplicações 3D que usam aceleração de hardware, por meio da abstração das principais APIs gráficas do mercado e também por meio do uso de abstrações de alto nível para lidar com a manipulação de cenas tridimensionais. A OGRE não é uma *engine* de jogos (ou *Game Engine*) e portanto não oferece funcionalidades para

[9]Uma cena é um termo que descreve um ambiente tridimensional. Uma cena pode conter sistemas de particulas, pontos de luz, malhas tridimensionais (ou *meshes*) entre outros elementos.

o uso de áudio, rede, IA, colisão entre outros. *Frameworks* específicos que implementam essas funcionalidades podem ser integrados com a OGRE para esse fim, e existem *engines* de jogos baseadas na OGRE que já oferecem integração como por exemplo *Monster Engine* e *Yake Engine*. Nesse projeto usaremos a OGRE diretamente porque estamos interessados somente na parte de abstração da complexidade de lidar com a renderização tridimensional.

A OGRE é uma engine que procura seguir os preceitos da Orientação a Objeto e Padrões de Projeto de *software* [28], e conta com uma coleção de classes para lidar com as diversas facetas que podem surgir na manipulação de cenas tridimensionais. O diagrama da figura 3.3 mostra algumas de suas principais classes.

No topo do diagrama da figura 3.3, temos a classe `Root` que tem a função de ser um objeto organizador que facilita o acesso aos diversos elementos da *engine*. É a partir do objeto `Root` que criamos outros objetos importantes como `SceneManager` e `RenderSystem` que serão explicados adiante. Também é a partir de `Root` que carregamos *plugins* e configurações da *engine* como por exemplo qual implementação de `RenderSystem` vamos usar (Direct3D ou OpenGL) ou qual vai ser a resolução da janela de renderização. `Root` deve obrigatoriamente ser o primeiro objeto OGRE a ser criado e deve ser o último a ser destruído pela aplicação cliente.

O objeto `Root` também nos permite definir a estratégia de renderização[10]. Na primeira estratégia, a aplicação cliente fica responsável por usar o método `renderOneFrame()`, que vai renderizar o estado atual da cena na janela de renderização representada pela classe `RenderWindow`. A aplicação cliente fica responsável por atualizar a cena e chamar esse método quando lhe for conveniente. Outra estratégia, é usar o método `startRendering()` que vai iniciar loop e um mecanismo de *callback*.

```
void Root::addFrameListener(FrameListener *listener);
virtual bool FrameListener::frameStarted(const FrameEvent &evt);
virtual bool FrameListener::frameEnded(const FrameEvent &evt);
```

Usando o método `addFrameListener` acima, podemos fornecer um *listener* que será chamado pelo objeto `Root` quando um novo quadro estiver pronto para ser renderizado. `Root` chama `frameStarted` quando um novo quadro começa a ser renderizado e chama o método `frameEnded`, quando a renderização do quadro termina. A aplicação cliente pode atualizar a cena por meio desses métodos. O

[10]Renderização é o processo pelo qual uma imagem em 2D é obtida a partir de uma cena ou modelo tridimensional. É um assunto complexo que possui extensa bibliografia. Citamos dois livros como referência, *Real-Time Rendering* [50] e *Computer Graphics: Principles and Practice in C* [26].

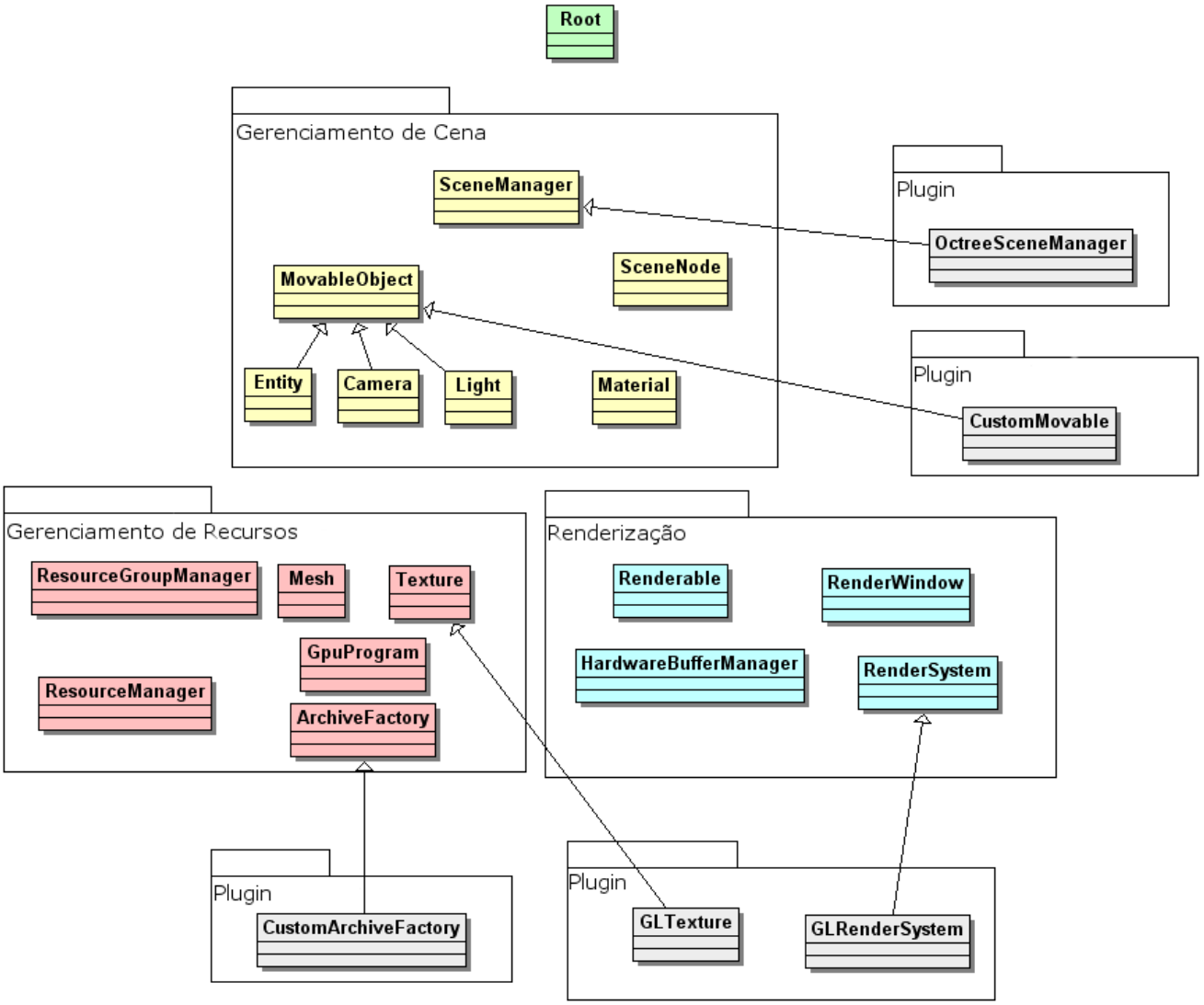

Figura 3.3: Diagrama mostrando algumas das principais classes da *engine* OGRE.

loop termina quando todas as janelas de renderização estiverem fechadas ou quando `frameStarted` ou `frameEnded` retornarem `false`.

Na figura 3.3 podemos ver que as classes estão divididas em quatro grupos: *Gerenciamento de Cena*, *Gerenciamento de recursos*, *Renderização* e *Plugins*. No grupo *Gerenciamento de Cena*,

econtramos classes que representam o conteúdo de uma cena tridimensional, definem como esse conteúdo está estruturado e definem como esse conteúdo é visto pelas câmeras[11]. As classes nesse grupo oferecem uma abstração das primitivas oferecidas pelas APIs de mais baixo nível como OpenGL e Direct3D, como triângulos, polígonos ou esferas. Podemos por exemplo, dizer para a OGRE instanciar um modelo tridimensional pronto e com materiais pré-definidos em lugar de construí-lo dinamicamente usando tais primitivas.

Gerenciamento de Recursos é um grupo de classes que contém funções para carregar recursos externos como texturas, fontes e modelos tridimensionais. O grupo *Renderização* contém classes cuja função é gerar uma imagem 2D que será mostrada na tela a partir da cena, abstraíndo os detalhes e a complexidade de como isso é feito. O pacote de instalação normal da OGRE vem com uma variedade de funcionalidades para atender as necessidades básicas de diversas aplicações. Para extender ou implementar novas funcionalidades, a OGRE tem uma estrutura de *plugins* que pode ser usada para esse fim.

A classe `RenderSystem` é um *Wrapper* para as APIs gráficas específicas que se comunicam com o *hardware*. `RenderSystem` é responsável por enviar os comandos para o hardware e configurar diversas opções de renderização. Normalmente, para as tarefas mais comuns não é necessário usar diretamente essa classe, já que a classe `SceneManager` oferece um maior nível de abstração.

A classe `SceneManager` é responsável por todos os elementos da cena que serão renderizados pela *engine*, o que significa que `SceneManager` acaba sendo a classe mais usada pela aplicação cliente nas interações com a *engine*. `SceneManager` permite a criação e o gerenciamento de objetos como `Camera`, `Entity` (que será explicado adiante) e `Light` que são os pontos de luz. Com essa classe, a aplicação cliente não precisa gerenciar a lista de objetos que está usando porque cada objeto criado tem um nome único que é designado pela aplicação cliente no momento de sua criação.

Diferents especializações de `SceneManager` existem para tratar dos diferentes tipos de cena que podem ser criadas em uma aplicação tridimensional. Cada tipo de cena pode apresentar diferentes abordagens para otimizar a renderização. Por exemplo, em cenas onde temos muitas paredes, podemos usar uma estrutura chamada de árvore BSP, *Binary Space Partitioning* [27]. Essa técnica é implementada em na classe `BspSceneManager`, que é uma especialização de `SceneManager`.

A classe `Mesh` representa um conjunto de polígonos ou primitivas geométricas tridimensionais

[11] Uma cena é renderizada a partir de um ponto de vista. Esse ponto de vista é representado pela classe *Camera* na *engine* OGRE.

que representam algum objeto que pode se deslocar na cena, como por exemplo, o modelo de uma pessoa. Um objeto `Mesh` é um tipo de recurso, designado pela extensão `.mesh`, e é carregado pela classe `MeshManager`, que é uma especialização da classe `ResourceManager`. O arquivo `.mesh` é obtido por meio da exportação de um modelo tridimensional em uma ferramenta de modelagem 3D, cujo plugin de exportação é disponibilizado pela OGRE. Cada ferramenta de modelagem 3D precisa de um plugin diferente e a OGRE procura suportar as principais ferramentas de modelagem do mercado. Um `Mesh` também pode ser animado. A animação é definida também na ferramenta de modelagem tridimensional, e exportada juntamente com o arquivo `.mesh`.

A classe `Entity` representa uma instância de um objeto que pode se mover em uma cena. Pode ser uma pessoa, um monstro, um robô ou um carro. Um objeto `Entity` sempre está relacionado a um objeto `Mesh` sendo que vários objetos `Entity` podem estar associados a um mesmo objeto `Mesh`. Por exemplo, podemos associar varios objetos `Entity` representando carros de polícia, a um mesmo objeto `Mesh` que contém os polígonos que efetivamente contituem o carro, assumindo que queremos várias carros de polícia iguais.

A criação de um objeto `Entity` acontece por meio do método `SceneManager::createEntity`, onde são passados como parâmetro o arquivo `.mesh` no qual a entidade será baseada e um nome para que a entidade possa ser posteriormente referenciada pela aplicação. Apenas uma cópia do mesmo objeto `Mesh` será carregada na memória.

Um objeto `Entity` não faz parte de uma cena até que tenha sido associado a um objeto `SceneNode`. Um objeto `SceneNode` pode ter várias entidades associadas. Esse objeto guarda as coordenadas e orientação da entidade na cena. Trata-se de um objeto lógico, porque ele não é representado na cena. Um exemplo do uso de `SceneNode` pode ser o seguinte: queremos que um ponto de luz acompanhe o modelo de uma pessoa em uma cena toda vez que essa pessoa se deslocar. Podemos implementar esse exemplo associando a entidade da pessoa e o ponto de luz no mesmo objeto `SceneNode`. Assim, ao deslocar o objeto `SceneNode` estaremos deslocando o modelo da pessoa e o ponto de luz ao mesmo tempo.

Um objeto `Mesh` pode ter vários `materiais` associados. Um material é representado pela classe `Material`. O material especifica propriedades das superfícies de um objeto como cor, reflexão de cores e texturas. Os materiais podem ser criados dinamicamente por programação, ou especificados em arquivos separados chamados de arquivos de *script*, que são carregados em tempo de execução juntamente com os objetos `Mesh`. Podemos dizer que um material define a aparência de um objeto

enquanto um objeto `Mesh` define sua forma.

Capítulo 4

Modelagem da Ferramenta

Nesse capítulo, vamos apresentar a modelagem da ferramenta. Nossa intenção foi colocar uma camada de abstração em cima das APIs da OGRE e do PROLOG, e tomar como base cenários de aplicação que acontecem em tabuleiros, como no caso do labirinto, o Mundo Wumpus, Xadrez, dentre outros. Deste modo, podemos aproveitar uma parte da estrutura para facilitar a criação de outros cenários.

4.1 O Conjunto de Classes da Ferramenta

Nosso objetivo com a ferramenta, é oferecer dois cenários sob a forma de problema, o Labirinto e o Mundo de Wumpus, e oferecer uma interface de forma que o aluno possa implementar a lógica que resolva esses problemas em PROLOG. É importante que essa ferramenta seja extensível, caso haja o interesse de implementar cenários adicionais. Como uma característica comum do cenário do Mundo de Wumpus e do Labirinto é uma espécie de quadriculado ou tabuleiro, fica mais natural pensar em cenários adicionais que possuam essa mesma característica, como por exempo Xadrez, Go[1] e o jogo de Damas.

Nessa linha, começamos apresentando a classe que implementa o tabuleiro, que chamamos de `DiscreteGrid`, a qual se encontra no diagrama de classes da figura 4.1. Essa classe representa um plano no espaço, dividido em quadrados como podemos ver na figura 4.2. O centro do plano π contido na classe `DiscreteGrid` contém a coordenada $(0, 0, 0)$ e π é paralelo ao plano XZ. O plano é divido em $n \times n$ quadrados de tamanho s, sendo que s e n são parâmetros de configuração de `DiscreteGrid`. Texturas podem ser aplicadas independentemente ao plano e aos quadrados, o

[1] Go é um jogo popular de tabuleiro muito antigo, originado na china [64].

que nos permite colocar informações visuais para o usuário. Por exemplo, podemos fazer com que o quadrado de coordenada $(0, 0)$ se torne um obstáculo e expressar isso visualmente trocando sua textura. Na figura 4.2 podemos ver um quadrado como uma textura diferente da textura dos outros quadrados.

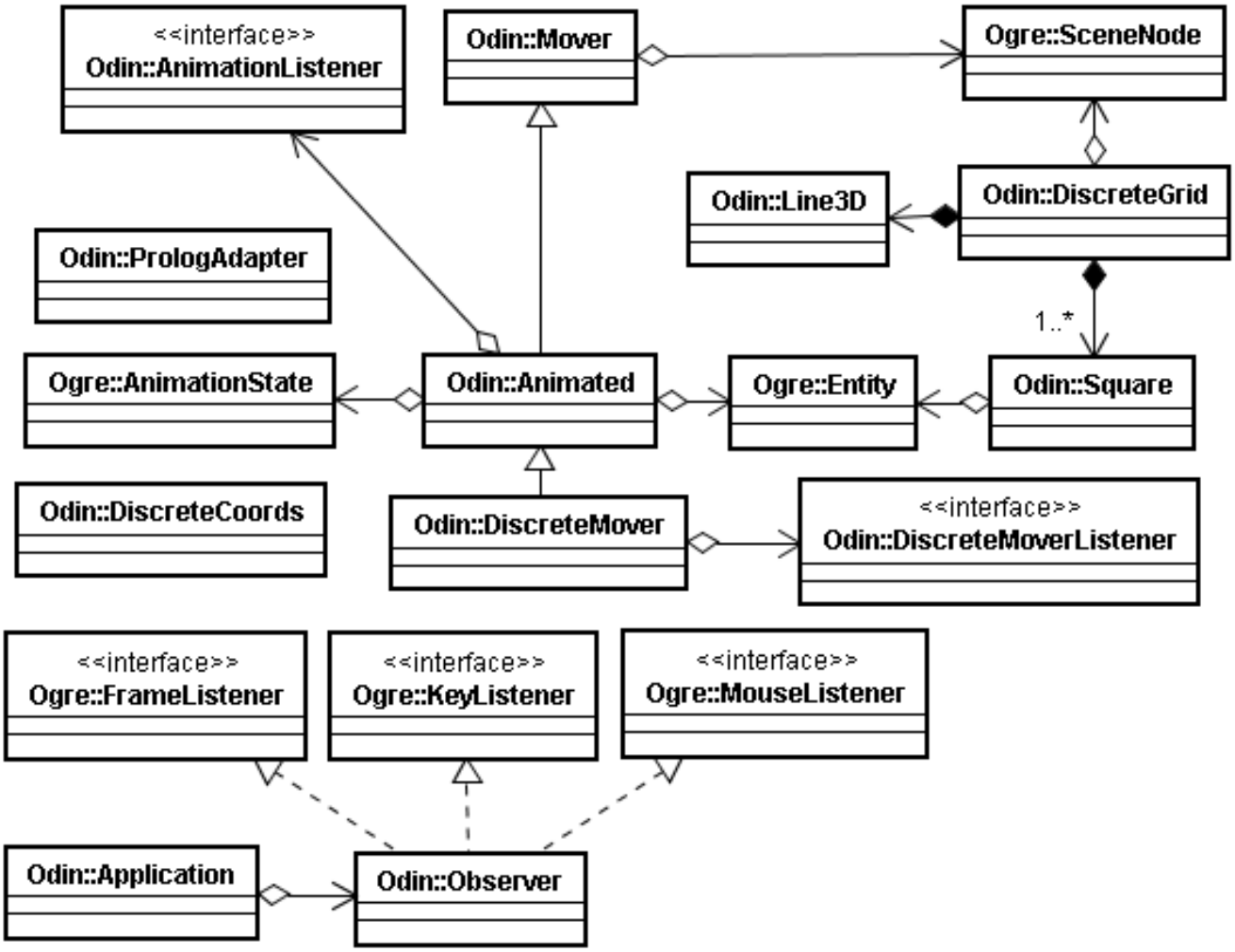

Figura 4.1: Diagrama de classes da ferramenta. Chamamos a ferramenta pelo codinome *Odin*, que usamos também para definir o *namespace*.

O plano de `DiscreteGrid` é um `Mesh` criado dinamicamente com uma textura associada. Tanto

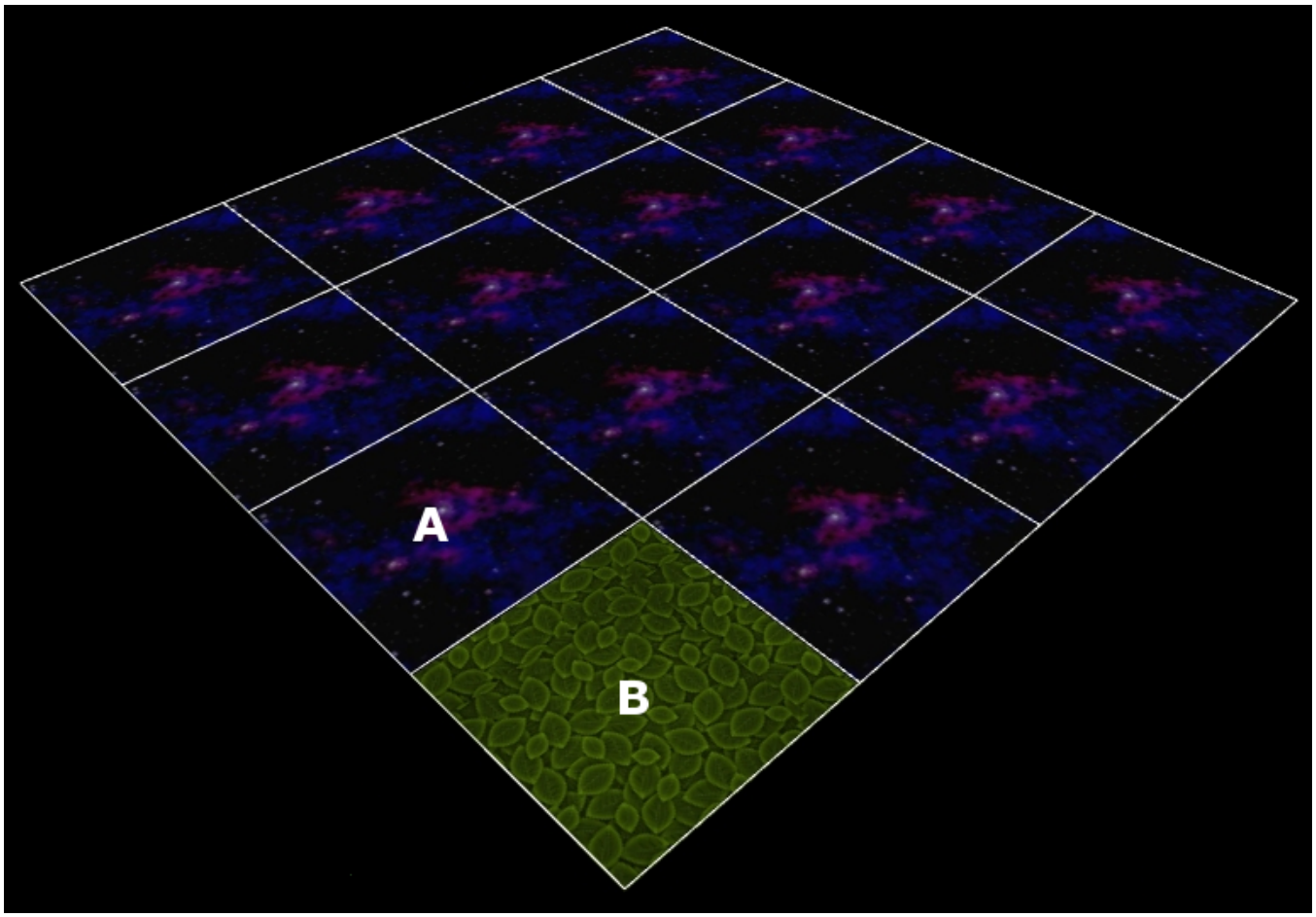

Figura 4.2: A classe DiscreteGrid sendo renderizada pela OGRE. As linhas que representam a divisão entre os quadrados podem ser escondidas. A textura do plano como podemos ver no quadrado A, e a textura dos quadrados individuais como podemos ver no quadrado B, podem ser configuradas dinamicamente.

esta textura como o tamanho do plano podem ser configurados como dito anteriormente. Os quadrados internos também estão associados com um `Mesh` e cada quadrado tem sua textura individual, como podemos ver na figura 4.2. `DiscreteGrid` mantém uma lista de quadrados em uma estrutura de Matriz, cujos atributos de textura e visibilidade podem ser manipulados pela aplicação cliente. A classe que representa os quadrados é chamada de `Square`. O efeito visual é obtido porque os objetos `Square` têm um componente de coordenada Y levemente superior que o componente de coordenada Y do plano. As linhas que representam a divisão dos quadrados são implementadas por meio ma-

nipulação do mecanismo de *Buffer* de vértices representado pela classe `HardwareBufferManager` da OGRE.

`DiscreteGrid` tem outra funcão importante. Em virtude de estarmos tratando o plano como um espaço discreto, é importante termos uma forma de, dado um ponto sobre o plano, pode saber qual quadrado o contém. Isso é útil por exemplo, para saber qual foi o quadrado clicado pelo usuário. Para resolver esse problema, a classe `DiscreteGrid` encontra o ponto no plano que foi clicado pelo usuário, por meio da utilização do método `getCameraToViewportRay()` da classe `Camera` da OGRE, e usa a classe `DiscreteCoords` para saber qual é o quadrado que contém esse ponto. A classe `DiscreteCoords` tem um método que retorna a coordenada do quadrado, a partir de um ponto qualquer no plano. Dada uma coordenada válida[2], podemos encontrar o `Square` que está nessa posição.

Vamos partir agora para as classes que representam os elementos que serão colocados em cima do tabuleiro. A classe `Mover` representa objetos que podem se mover no espaço. `Mover` oferece diversos métodos voltados para o movimento e a rotação de um objeto. É também na classe `Mover` que fica a informação da posição do objeto móvel, porque `Mover` tem uma referência para o objeto `SceneNode` como podemos ver na figura 4.1.

Com o objetivo de conseguir um maior detalhe visual, vamos considerar que os objetos móveis podem ser animados. A classe `Animated`, no diagrama da figura 4.1, encapsula as operações de animação oferecidas pela OGRE. Para um objeto ser animado, não basta termos apenas o *Mesh*, mas precisamos também das animações que se encontram em um arquivo separado com a extensão `.skeleton`. Esse arquivo define animações para o modelo que se encontra no arquivo `.mesh` correspondente. Cada animação pode ser referenciada por uma *string* que a identifica, como por exemplo "*idle*" ou "*walking*".

Estamos assumindo que todo objeto animado pode se mover, então a classe `Animated` é uma especialização de `Mover`. A classe `Animated` tem uma referência para o objeto `Entity`, e uma referência para a animação atual, representada pelo objeto da OGRE `AnimationState`. Por meio de `Animated` podemos trocar a animação dinamicamente e é assim que implementamos por exemplo, a troca da animação de "caminhada" para a animação de "repouso", quando um agente chega em seu destino.

Podemos ver na figura 4.1 que `Animated` possui uma referência para um objeto `AnimationListener`.

[2]Por exemplo, se o plano tem 2×2 quadrados, as coordenadas válidas são $(0, 0)$, $(0, 1)$, $(1, 0)$ e $(1, 1)$. As coordenadas $(-1, 0)$ e $(2, 1)$ seriam exemplos de coordenadas inválidas.

`Animated` invoca um método em `AnimationListener` quando uma animação que se não se repete chega ao fim. Uma animação pode ser configurada tanto para se repetir[3] ao chegar no final, como para não se repetir. Uma animação de caminhada por exemplo, é configurada para se repetir, porque o caminho a ser percorrido pode ser arbitrariamente longo, enquanto a animação tem um número de quadros definido. Já uma animação de um golpe por exemplo, normalmente não é repetida, porque trata-se de um movimento pontual bem definido. Nesses casos podemos querer realizar algum processamento no momento em que um movimento terminar. Por exemplo, podemos querer que depois que o agente desferir um golpe contra outro agente, uma quantidade de energia seja descontada imediatamente do agente golpeado. É por isso que a classe `Animated` oferece um meio para registrar um objeto `AnimationListener`.

O fato de termos um espaço discretizado, implica que os objetos devem se mover sobre esse espaço de maneira bem definida. Vamos definir que um agente deve se mover de um quadrado A até outro quadrado B, partindo do centro de A e chegando ao centro de B, para termos a impressão visual de que os quadrados fazem sentido e que seus limites estão sendo respeitados. A classe `DiscreteMover` implementa um objeto que sabe se mover no plano de `DiscreteGrid`, utilizando a classe `DiscreteCoords` para obter os centros dos quadrados, por meio de um método que, dada uma coordenada de um quadrado, retorna um ponto em $\Re^3$ com a posição do centro do quadrado.

`DiscreteMover` deve ser configurada com duas animações: a animação do objeto em repouso, e a animação do objeto quando ele está em movimento. Podemos fazer com que o objeto comece a se mover para o quadrado escolhido por meio do seguinte método:

```
void moveToSquare(const DiscreteCoords &coords);
```

O parâmetro `coords` precisa ser um quadrado válido. Uma regra adicional é que o objeto `DiscreteMover` só pode se mover de um quadrado, para um outro quadrado que esteja adjacente.

Para atualizar a animação de `DiscreteMover`, usamos o seguinte método:

```
virtual void move(Real time);
```

Esse método normalmente é invocado a partir do *loop* principal de animação, como veremos adiante. O parâmetro `time`, indica o tempo decorrido desde a renderização do último quadro (ou

[3] também podemos dizer "animação em *loop*".

Frame). O método `move` também atualiza o estado do objeto `DiscreteMover`, caso o objeto parta do repouso, ou caso o objeto chegue ao seu destino. No primeiro caso, a animação do objeto em repouso é trocada pela animação do objeto em movimento e vice-versa no segundo caso.

Assim como a classe `Animated`, `DiscreteMover` também possibilita a adição de um *listener* representado pela classe `DiscreteMoverListener`, como podemos ver na figura 4.1. O *listener* é notificado no momento em que o objeto `DiscreteMover` chega em seu destino. Isso é útil porque podemos precisar tomar alguma ação nesse exato momento. Por exemplo, suponha que o quadrado no qual o agente acabou de chegar é uma mina, e que queremos explodi-la no exato momento em que o agente chega nesse quadrado. Para fazer isso, registramos um objeto `DiscreteMoverListener` que irá explodir a mina quando acionado.

O principal ponto de extensão disponibilizado pela estrutura de classes da ferramenta é a classe `Observer`, que é um *listener* que concentra os eventos de entrada de dados de *mouse* e teclado, bem como eventos de renderização. `Observer` implementa as interfaces *listener* da OGRE, `FrameListener`, `MouseListener` e `KeyListener` como podemos ver na figura 4.1. Depois que `Observer` é registrada como *listener* dos eventos na classe da OGRE, `EventProcessor`, os eventos começam a ser recebidos. Isso acontece logo no começo da execução da ferramenta, com a criação da classe da ferramenta `Application`, como veremos adiante.

A classe `Observer` contém o método que concentra o *loop* principal do aplicativo. O método se chama `frameAction`, cuja assinatura é mostrada abaixo:

```
virtual bool frameAction(const FrameEvent &evt);
```

É nesse método que normalmente é implementada a atualização da simulação. Caso existam por exemplo, quaisquer objetos `DiscreteMover`, o método `move()` de `DiscreteMover` será invocado aqui. O parâmetro `evt` do método `frameAction` acima, tem um atributo que indica quantos miliseguindos se passaram desde que o último *frame* foi renderizado. Esse número pode então ser usado para atualizar a animação e o movimento dos objetos.

`Observer` possui um método importante, responsável por inicializar objetos OGRE, e tem a seguinte assinatura:

```
virtual void init();
```

Esse método é invocado pela classe `Application` no início da execução da simulação. É necessário fazer isso no lugar de usar o construtor de `Observer` porque `Application` recebe uma referência de um objeto `Observer` já criado. `Application` contém o código de incialização da *engine* OGRE, e portanto, como `Observer` pode conter objetos OGRE, inicializar esses objetos OGRE antes da inicialização da própria *engine* acaba derrubando o programa. Portanto, a inicialização de objetos OGRE, ou objetos associados a um objeto OGRE deve ser necessariamente realizada no método `init()` da classe `Observer` e nunca em seu construtor.

A classe `Observer` implementa por padrão a navegação no ambiente. Usando as teclas `A,D,S,W`, o usuário pode se mover para os lados e para frente ou para atrás respectivamente. Com o mouse, o usuário gira a câmera para poder olhar ao seu redor.

O código da função *main* de um cenário fica da seguinte maneira:

```
void main() {
    PlEngine e("");

    Observer *observer;
    Application *application;

    observer = new ObserverImpl();
    application = new Application(observer);
    application->go();

    delete observer;
    delete application;
}
```

No trecho acima, a classe `ObserverImpl` é uma especialização da classe `Observer`. Diferentes cenários terão implementações diferentes de `Observer`. No caso do cenário do labirinto por exemplo, precisamos de um ou mais agentes que saibam se mover pelo espaço encontrando o melhor caminho, e também precisamos criar obstáculos no plano, representando as paredes do labirinto. Já no caso do Mundo de Wumpus, precisamos de um agente que fará o papel do aventureiro, e precisamos de um monstro representando o Wumpus, entre outros requisitos como veremos adiante. Toda essa lógica é

implementada em uma especialização da classe `Observer`. O objeto `Observer` é então passado para o objeto `Application`, como podemos ver no código acima, e então o *loop* principal é iniciado na chamada do método `go()`. A partir desse momento, a OGRE vai renderizar a simulação *frame* a *frame*, e a classe `Observer` vai ser acionada por meio método `frameAction` no começo da renderização de um *frame*.

4.2 Cenário do Labirinto em Funcionamento

Nesta seção vamos ver como funciona o cenário do labirinto e quais são as interfaces que o usuário deve conhecer para escrever o código em PROLOG que resolve o problema do labirinto.

Esse cenário propõe escrever um código que resolva o problema de, dadas duas coordenadas no tabuleiro, achar o menor caminho entre essas coordenadas se existir, considerando que podem existir obstáculos em coordenadas arbitrárias em todo o tabuleiro.

No tabuleiro, a simulação será representada por um agente que obedece comandos do usuário. Inicialmente, o agente se encontra em um quadrado A. O usuário então clica em um quadrado B, e a tarefa do agente será encontrar e percorrer o caminho $\overline{AB}$, que deve ser o menor possível. O usuário também pode criar obstáculos pressionado a tecla `ctrl` e clicando em um quadrado arbitrário no tabuleiro. O caminho $\overline{AB}$ é a lista de quadrados que deve ser percorrida em ordem para que o agente possa ir de A a B.

Quando o usuário clica em quadrado indicando um destino para o agente, a ferramenta tenta unificar o predicado `findPath(+,+,-)`, que deve estar escrito em um arquivo `.pl` e armazenado no diretório `deploy` da ferramenta. O predicado tem o seguinte formato:

```
findPath(From, Goal, Path).
```

O argumento `From`, indica a coordenada em que o agente está, enquanto `Goal`, é a coordenada de destino. O argumento `Path` é uma lista de direções que o agente deve seguir para chegar ao destino. A partir de um quadrado, o agente pode tomar 8 direções: as adjacências do quadrado, e as diagonais. As direções são representadas pelas *strings* $d1, d2, ..., d8$. A string $d1$, é a direção positiva ao longo do eixo Z em relação ao sistema de coordenadas, e assim segue no sentido horário até $d8$. Uma possível lista de direções que poderia ser unificada em `Path` é $[d1, d1, d2]$ por exemplo. Os argumentos `From` e `Goal` indicam uma posição de um quadrado. A posição de um quadrado é

representada pelo predicado `node(X,Y)` com `mode node(+,+)`. Uma implementação extremamente simples poderia ser a seguinte:

```
findPath(_,_,[d1]).
```

Ou seja, estamos dizendo que para quaisquer pontos de origem e destino, queremos que o agente ande sempre na direção indicada pela *string* $d1$. Podemos escrever algoritmos mais elaborados como veremos a seguir.

O código em PROLOG tem acesso ao estado de todos os quadrados no tabuleiro no momento do cálculo do caminho, ou seja, não há um raio de visão para o agente. Para verificar se uma dada posição é um obstáculo ou não, usamos o predicado `obstacle(+)`:

```
obstacle(node(X,Y)).
```

Se `node(X,Y)` corresponder à posição de um obstáculo, então o predicado `obstacle(+)` é unificado pelo PROLOG. Dessa forma, pode-se escrever um código que encontre um caminho que se desvia de eventuais obstáculos.

A ferramenta Odin usa o predicado `assert(+)` para colocar os obstáculos criados pelo usuário na interface 3D no momento do clique. Na figura 4.3, podemos ver como fica o cenário do labirinto na ferramenta. O caminho escolhido pelo agente é mostrado visualmente na ferramenta, como podemos ver na figura 4.4.

Naturalmente, existe a possibilidade de que o código gere um caminho inválido, ou que não considere os obstáculos. No caso de um caminho inválido, ou no caso de existir algum erro de sintaxe PROLOG, o programa simplesmente termina, indicando o erro em uma caixa de diálogo. No evento de ser retornado um caminho que não considere obstáculos, gerando possivelmente uma colisão, a Odin vai mostrar visualmente o caminho e indicar o ponto da colisão, como podemos ver na figura 4.5. O algoritmo escrito em PROLOG usado nesse caso é baseado na Distância de Manhattan[4] e pode ser encontrado na subseção A.2.1 do apêndice.

[4]Resumidamente, a Distância de Manhattan é a distância entre dois pontos medida ao longo dos eixos em ângulos retos, frequentemente aplicada em espaços com aspectos discretos como é o caso do cenário do labirinto. A Distância de Manhattan pode ser calculada da seguinte maneira: seja p um ponto no plano π na coordenada (x, y) e p' outro ponto no plano π na coordenada (x', y'), a distância $\overline{pp'}$ no plano π é dada por $|x - x'| + |y - y'|$. Mais informações sobre a distância de Manhattan podem ser encontradas em [41].

Figura 4.3: Cenário do Labirinto da ferramenta Odin. O agente é representado pelo robô. Os obstáculos, são representadas pelos discos amarelos.

Em constraste, na figura 4.6 podemos ver um exemplo de um algoritmo que leva os obstáculos em consideração na geração do caminho. O algoritmo usado nesse caso foi o algoritmo A-estrela[5], que é um algoritmo de busca em grafos que usa uma função heurística $f(x)$ que estima o comprimento da melhor rota que passa pelo vértice x. Os vértices com a menor estimativa são analisados primeiro. O algoritmo foi descrito em 1968 por Hart, Nilsson e Raphael em [30]. A implementação em PROLOG

[5]Também chamado de A*.

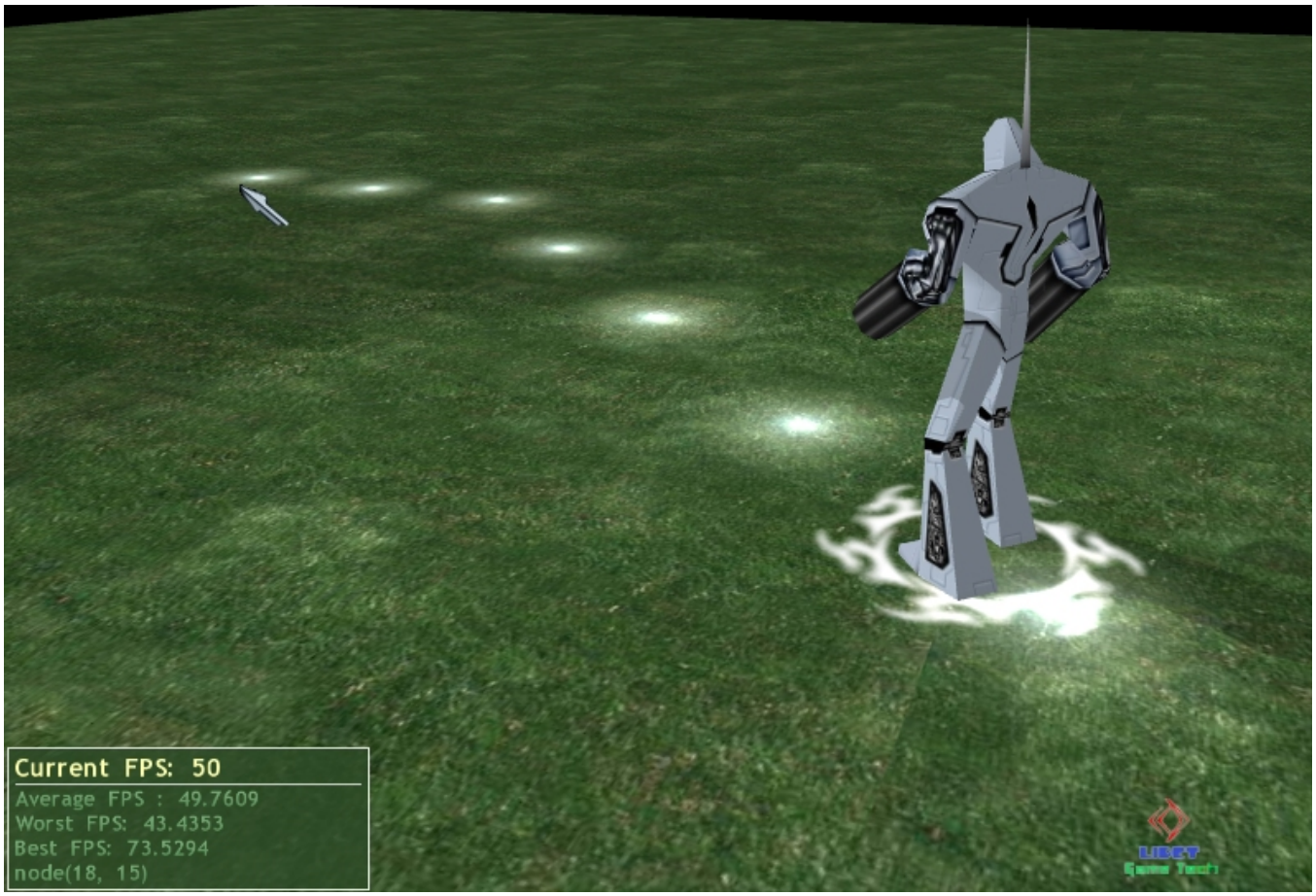

Figura 4.4: O caminho válido escolhido pelo agente pode ser visualizado pelos pontos brilhantes no plano.

pode ser encontrada na seção A.2.2 do apêndice.

Para implementar o cenário do labirinto escrevemos especializações das seguintes classes: a classe `DiscreteGrid` foi extendida para encapsular as operações de mudança de textura dos quadrados do tabuleiro, como criar um obstáculo e indicar visualmente o caminho escolhido. A classe `DiscreteMover` foi especializada para receber uma lista de quadrados a serem visitados em ordem. Finalmente, foi escrita uma especialização de `Observer` para receber os comandos de mouse e teclado e orquestrar a aplicação.

Figura 4.5: Nessa figura, o caminho escolhido pelo agente atravessa um obstáculo.

4.3 Cenário do Mundo de Wumpus em Funcionamento

O cenário do Wumpus propõe escrever o código em PROLOG que resolva o problema do Mundo de Wumpus, descrito na seção 3.2.2. O problema do cenário do Mundo de Wumpus consiste em achar uma pepita de ouro que está em alguma parte do tabuleiro que representa uma caverna, voltando posteriormente para a posição inicial e finalmente sair da caverna por meio de uma escalada. No mundo de Wumpus, o agente não conhece o estado dos quadrados do tabuleiro, ele apenas conhece o estado do quadrado em que se encontra no momento, sendo que cabe ao agente guardar informações sobre quadrados já visitados. No cenário do Mundo de Wumpus estamos usando as seguintes regras:

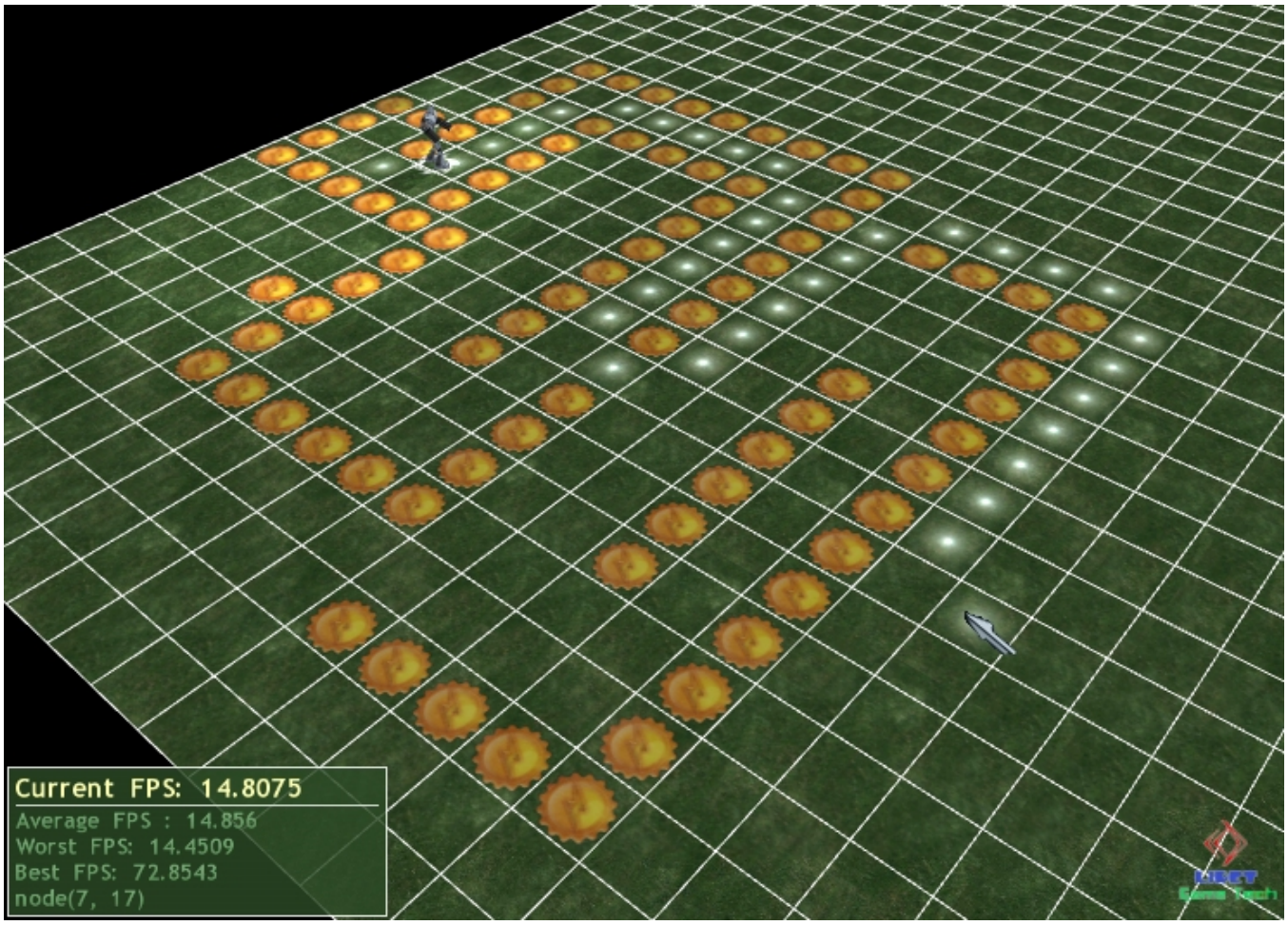

Figura 4.6: Nessa figura, os obstáculos foram criados à semelhança de um labirinto. Mesmo assim o agente consegue achar o caminho para o ponto escolhido.

- Só pode haver uma pepita de ouro em todo o tabuleiro.
- Pode haver um ou mais abismos no tabuleiro.
- O tamanho do tabuleiro pode ser acessado a partir do código PROLOG.
- O agente tem um golpe para matar o Wumpus. Este golpe pode ser desferido a partir de um quadrado adjacente ao quadrado em que o Wumpus se encontra e o agente deve estar de frente para o Wumpus. O agente pode usar esse golpe uma única vez na simulação.

- Nos quadrados adjacentes a um abismo, o agente pode sentir uma leve brisa.
- Nos quadrados adjacentes ao quadrado em que o Wumpus se encontra, o agente pode perceber um odor característico.
- O odor característico desaparece quando o Wumpus é executado pelo agente.

A simulação vai funcionar da seguinte maneira: um agente estará em uma posição inicial no tabuleiro, que pode ser configurada. A posição dos abismos, da pepita de ouro e do Wumpus também pode ser alterada em um arquivo de configuração. A cada vez que o usuário pressiona a barra de espaço do teclado, a Odin vai unificar um predicado que vai retornar uma *string* representando uma ação a ser tomada pelo agente. O agente pode escolher realizar uma dentre as seguintes ações:

- *climb*: o agente decide sair da caverna por meio de uma escalada. Só funciona na posição inicial.
- *punch*: o agente desfere um golpe mortal. Pode ser usado uma única vez e só funciona se o agente estiver de frente para o Wumpus e em um quadrado adjacente.
- *pick*: o agente tenta coletar a pepita de ouro. A ação será executada com sucesso se o agente estiver no mesmo quadrado que a pepita de ouro.
- *move*: move-se um quadrado para a frente, ou seja na direção em que esteja olhando no momento.
- *right*: o agente rotaciona seu corpo para a sua direita, 90°.
- *left*: o agente rotaciona seu corpo para a sua esquerda, 90°.
- *back*: o agente rotaciona seu corpo 180°.

A ferramenta Odin tentará unificar o predicado `act(-)`:

```
act(Action).
```

Uma possível, simples e válida implementação, que não terá sucesso, pode ser que a cada iteração o agente se moverá um quadrado para a frente:

```
act(move).
```

Nessa implementação estamos dizendo que a cada iteração, o agente se moverá para a frente. Vamos ver agora como fica o cenário na ferramenta. Na figura 4.7 podemos ver o agente e o tabuleiro e podemos perceber que somente o quadrado em que o agente se encontra no momento está revelado.

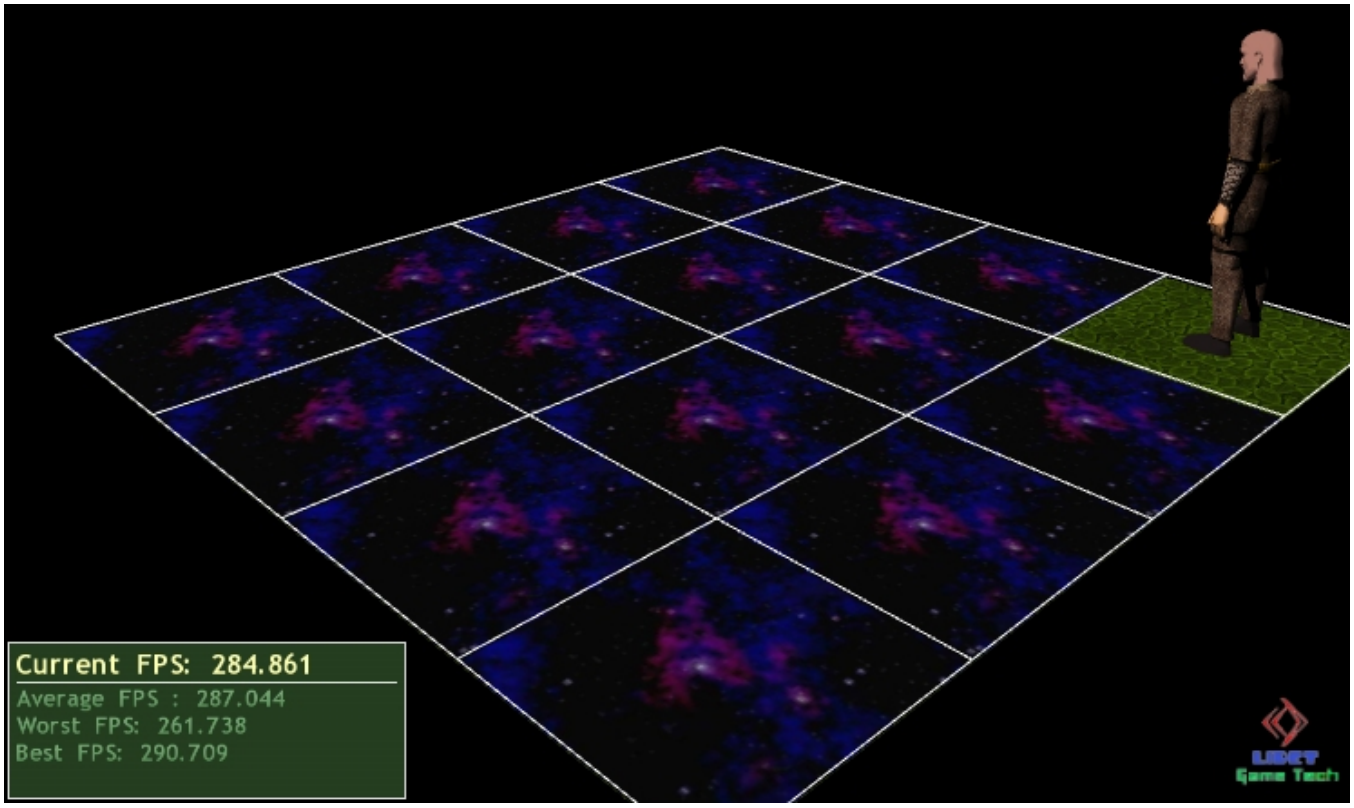

Figura 4.7: O agente e o tabuleiro 4×4. Somente o quadrado no qual o agente se encontra está revelado.

Na figura 4.8 podemos ver os sinais de perigo indicados visualmente pela ferramenta, depois que o agente já percorreu uma parte do cenário.

A qualquer momento, o usuário pode pressionar a tecla F do teclado para revelar todos os quadrados do tabuleiro. Isso é útil para fins de depuração do código, e para se ter uma idéia melhor do que está acontecendo no cenário. Na figura 4.9, podemos ver o cenário por completo, com o Wumpus, e os abismos.

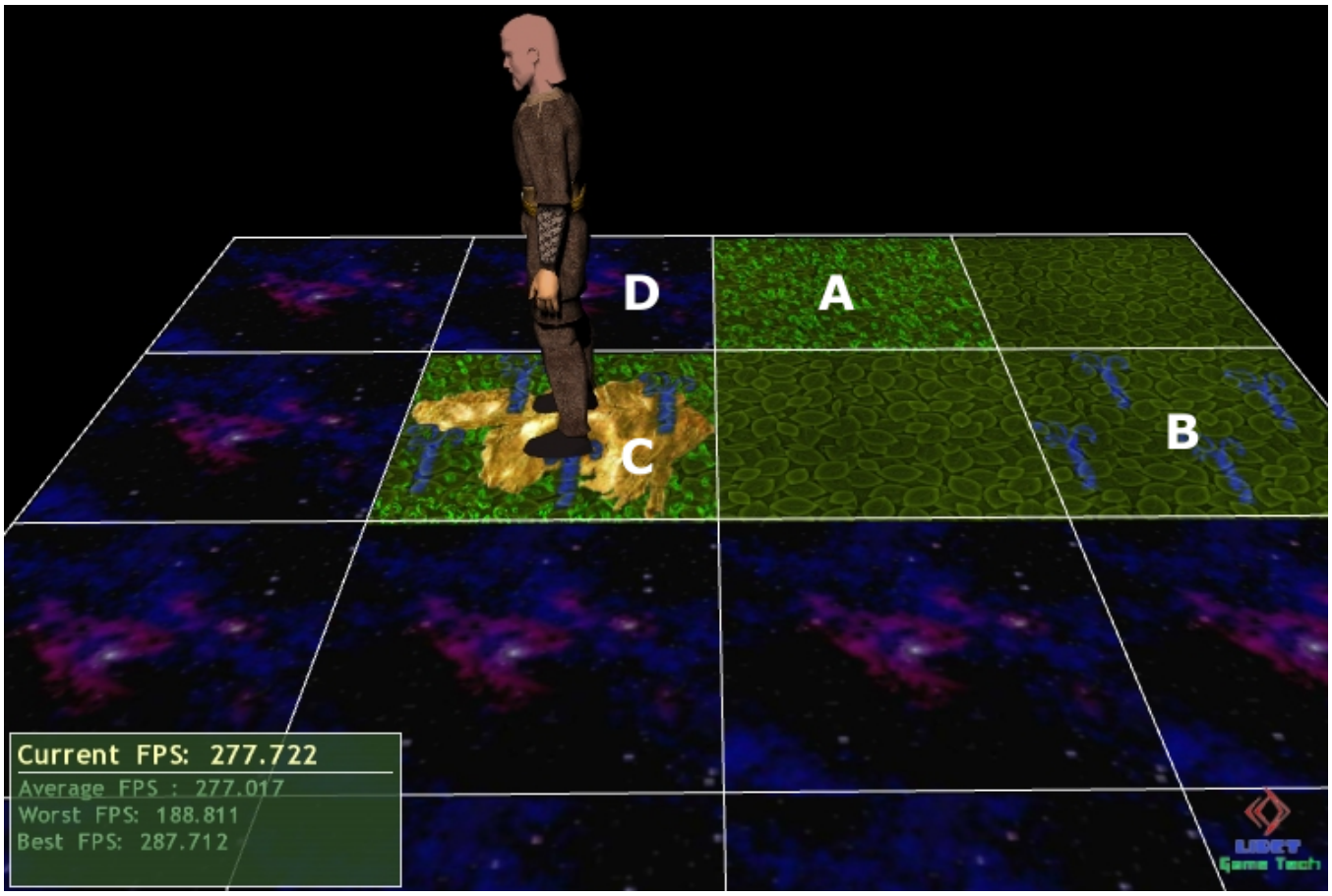

Figura 4.8: Os sinais de perigo que podem ser percebidos em B, a brisa indicando que um abismo está próximo, e em A o odor representado pela substância esverdeada, que indica que o Wumpus está próximo. O agente se encontra sobre a pepita de ouro em C. Nesse momento, o agente já tem condições de deduzir qual é o quadrado onde o wumpus se encontra, em função dos quadrados visitados em que se pode perceber o odor. O Wumpus se encontra no quadrado indicado por D.

No código PROLOG, o usuário tem acesso a uma série de predicados para analisar a situação atual do agente no tabuleiro. O primeiro predicado é `adventurer_orientation(-)`, que revela qual é a direção atual do agente no cenário:

```
adventurer_orientation(Orientation).
```

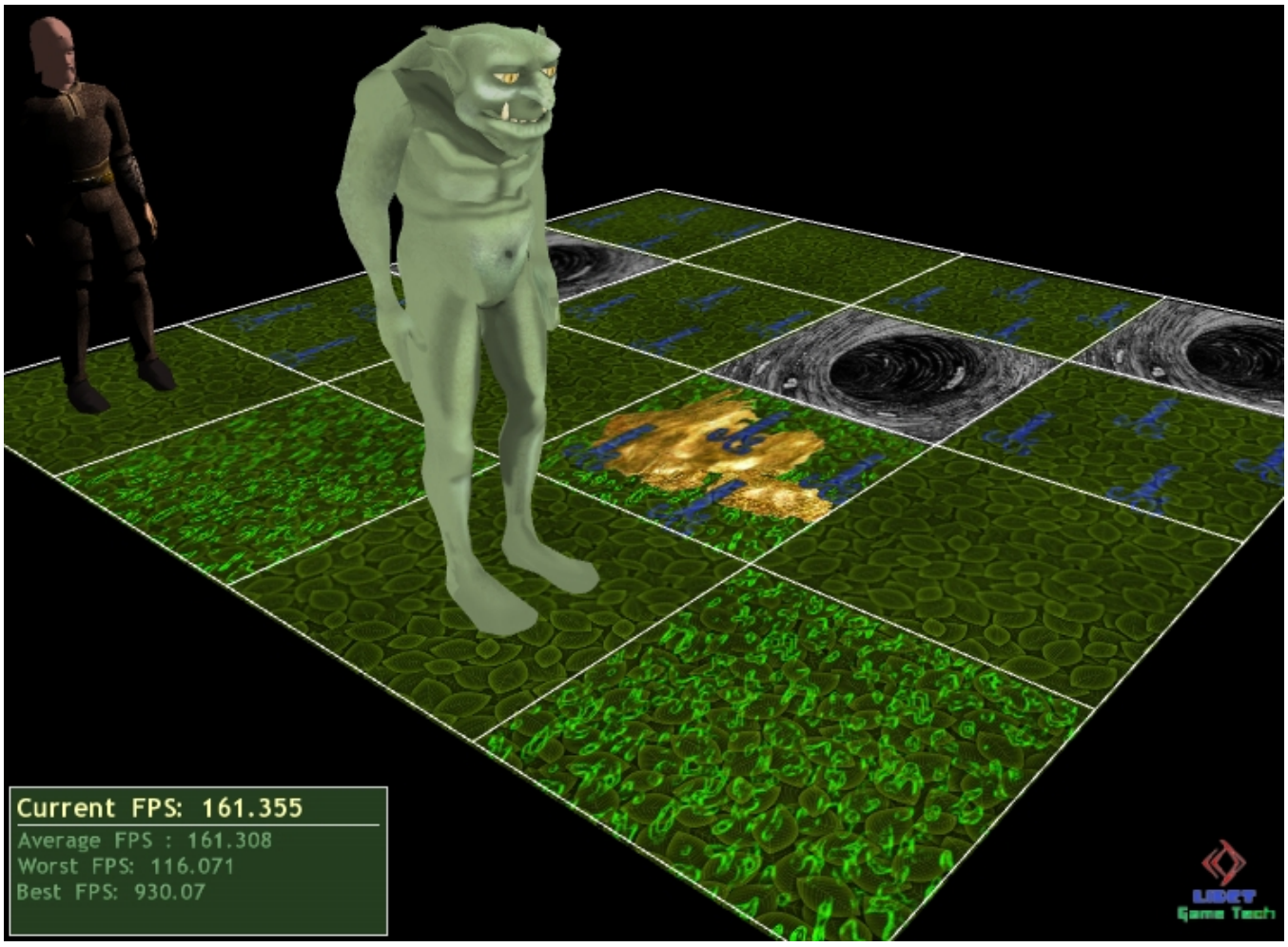

Figura 4.9: O tabuleiro revelado por completo.

A ferramenta Odin unifica o predicado `assert(+)` para tornar disponível o fato `adventurer_orientation`. O argumento `Orientation` é unificado com uma das seguintes *strings*: $d1, d2, d3, d4$. Essas *strings* significam 0°, 90°, 180° e 270° respectivamente, em relação ao eixo X do sistema de coordenadas. Outro predicado, `adventurer_location(-,-)`, indica qual é a posição atual do agente no tabuleiro:

```
adventurer_location(X,Y).
```

Os argumentos `X` e `Y` são números inteiros que indicam qual é a coordenada (x, y) do quadrado no tabuleiro.

O predicado `sideSquareNumber(-)` unifica um inteiro que retorna o número n do tabuleiro $n \times n$. Já o predicado `canPunch(-)` unifica a *string yes* caso o agente não tenha utilizado seu golpe ainda, e unifica a *string no* caso do agente já ter usado o golpe.

Por fim temos o predicado que faz o papel do sensor do agente, que é o predicado `percepts(-,-,-,-,-)`:

```
percepts(Stench, Breeze, Gold, Bump, Scream).
```

Todos os argumentos do predicado `percepts(-,-,-,-,-)` unificam com as *strings yes* e *no*, representando o valor booleano. Se o argumento `Stench` unifica com *yes*, então significa o wumpus se encontra em um quadrado adjacente, e consequentemente o agente se encontra sobre um quadrado com a substância esverdeada como mostra a figura 4.8. O argumento `Breeze` indica a existência ou não de um abismo em um quadrado adjacente. Se `Breeze` unificar com `yes`, então existe um abismo no quadrado adjacente, e unifica com `no` em caso contrário. O argumento `Gold` unifica com `yes` caso o agente se encontre sobre um quadrado que contenha a pepita de ouro e unifica com `no` em caso contrário. O argumento `Bump` unifica com `yes` caso o agente tenha tentado atravessar uma das quatro paredes do cenário e unifica com `no` caso contrário. Finalmente, o argumento `Scream` unifica com `yes` caso o Wumpus tenha sido morto, e consequentemente, emitido um estrondoso grito, e unifica com `no` caso contrário.

Um exemplo de algoritmo codificado em PROLOG para esse problema pode ser encontrado na seção A.3 do apêndice.

Para implementar o cenário do Mundo de Wumpus escrevemos especializações das seguintes classes: a classe `DiscreteGrid` foi extendida para encapsular as operações de mudança da textura dos quadrado: de revelado para não revelado, tornar o quadrado uma brisa, colocar a substância esverdeada nas adjacências do Wumpus, colocar uma pepita de ouro no quadrado e colocar um abismo no quadrado, como podemos ver na figura 4.9. Podemos observer na figura que é possível ter uma composição destes atributos, como por exemplo, em um mesmo quadrado podemos ter uma pepita de ouro, uma brisa e a textura representando o odor do Wumpus, cada combinação possível foi implementada como uma textura diferente.

A classe `DiscreteMover` foi especializada para encapsular todas as mudanças de animação do agente com por exemplo, a troca de animação de repouso para a animação de andar, a animação do soco e a animação da coleta da pepita de ouro. Finalmente, foi escrita uma especialização de `Observer` para receber os comandos de mouse e teclado e orquestrar a aplicação e carregar os arquivos de configuração específicos do cenário.

Capítulo 5

Considerações Finais

Neste trabalho, chamamos atenção para o fato de que no ensino de Computação, os jogos representam um ambiente virtual onde os alunos podem construir seus programas e visualizar seu comportamento. Também vimos que a visualização dos programas em funcionamento parece ter o efeito de prender a atenção do aluno. Mostramos alguns trabalhos realizados na direção de utilizar jogos no ensino de ciência da computação e que reportam que a abordagem é eficaz e encontra uma resposta positiva por parte dos alunos. Nessa linha, focamos nossas contribuições no Ensino de Inteligência Artificial, especificamente no ensino de lógica, onde aplicamos a idéia de visualizar programas PROLOG em funcionamento em cenários bem definidos.

5.1 Contribuições

Nossa principal contribuição nesse sentido foi o desenvolvimento de uma ferramenta que permite a visualização de especificações de comportamento de agentes em linguagem PROLOG. Nosso objetivo é fazer com que a ferramenta se torne um recurso didático interessante focado em cursos de lógica para graduação.

Com a ferramenta, o aluno não precisa se preocupar em escrever toda a camada de apresentação de seu algoritmo, porque a ferramenta já se encarrega disso para os cenários apresentados. Assim, o aluno só precisa se concentrar na lógica para resolver o problema. Além disso, a visualização dos programas é o que parece prender a atenção dos alunos, fazendo com que a ferramenta se torne uma ajuda para o professor nesse sentido.

Disponibilizamos dois cenários de aplicação. No cenário do Labirinto, o aluno deve resolver o problema de como achar o menor caminho entre dois pontos em um grafo. No cenário do Mundo

de Wumpus, o desafio é escrever uma lógica para um agente cujo objetivo é entrar em uma caverna, buscar uma pepita de ouro e voltar para a posição inicial, desviando-se de abismos e do monstro Wumpus no caminho.

O cenário do labirinto pode ser usado para visualizar os algoritmos da primeira parte do livro de Russel e Norvig [62], um texto bastante utilizado em cursos de IA. O cenário do Mundo de Wumpus é usado pelos autores do livro para ilustrar uma grande quantidade de temas de IA apresentados nos capítulos posteriores. Todas essas ilustrações podem ser visualizadas e estudadas através da ferramenta.

5.2 Uso da Ferramenta no IME/USP

A ferramenta foi utilizada em 2 cursos do Instituto de Matemática e Estatística da Universidade de São Paulo, IME/USP: Métodos Formais em Programação e Laboratório de Inteligência Artificial. Enquanto o objetivo não foi usar um ferramental estatístico e nem investigar questões de causalidade quanto à efetividade da ferramenta, a experiência serviu seu propósito de verificar empiricamente a reação dos alunos a essa abordagem.

Para analisar mais cuidadosamente a eficácia dessa ferramenta, mais experimentos precisam ainda ser efetuados, partindo de uma base mais significativa de alunos e usando ferramental estatístico apropriado.

5.3 Apresentação em Eventos

Citamos que durante o desenvolvimento do projeto, tivemos a oportunidade de publicar trabalhos nas edições 2005 e 2006 do Simpósio Brasileiro de Jogos de Computador e entretenimento digital [21, 22]. A ferramenta também foi apresentada em um evento acadêmico no México, voltado primordialmente a docentes do ensino superior da América Latina. O programa gerou grande interesse durante todo o evento.

5.4 Trabalhos Futuros

Uma continuação natural deste trabalho, seria, como já citado anteriormente, utilizar um ferramental estatístico e pedagógico apropriado para verificar se existe correlação entre o uso da ferramenta e o aprendizado ou nível de motivação dos alunos em cursos de lógica e IA. Acreditamos que além das disciplinas já citadas, as seguintes disciplinas também poderiam ser alvo do uso da ferramenta

em experimentos:

- MAC0425 - Inteligência Artificial, cujo objetivo é expor o aluno às diversas áreas da Inteligência Artificial, com aprofundamento em alguns tópicos como agentes com capacidade de resolução de problemas, percepção, planejamento e aprendizagem.
- MAC5784 - Inteligência Artificial em Jogos de Computador, cujo objetivo é apresentar os conceitos, técnicas e métodos de programação relacionados à área de Inteligência Artificial específicos para uso em jogos de computador.

Outro ponto de extensão interessante, seria criar mais cenários de aplicação estendendo o conjunto de classes base da ferramenta. Como sugestões podemos citar o xadrez, o Go [64] e o jogo de damas.

Baseando-se na idéia do projeto MUPPETS, de implementar um editor de texto Java dentro da ferramenta, uma sugestão é implementar um editor de texto PROLOG com funcionalidades básicas, para que o aluno não precise sair da ferramenta quando quiser fazer alterações no código.

A ferramenta Odin, pelo fato de embutir um interpretador PROLOG, é sujeita a entrar em *loop* infinito caso o programa PROLOG submetido pelo aluno contenha um *loop* infinito, fazendo com que a interface congele. Isso pode acontecer até mesmo em casos onde não exista um *loop* infinito, mas sim um trecho de código que leve um tempo excessivamente grande para ser executado. Uma melhoria que poderia ser feita nesse caso, seria executar as rotinas de integração com o PROLOG em uma *Thread* separada, de forma que a interface não congele nesses casos, dando a opção para o usuário de cancelar a operação.

Apêndice A

Apêndice

A.1 Código da classe PrologAdapter

Abaixo mostramos o código da classe `PrologAdapter`, usado pela ferramenta para fazer chamadas ao SWI-PROLOG partindo do C++.

```
#include <SWI-cpp.h>

#include <OdinHelperKit.h>
#include <OdinException.h>

#include <OgreNoMemoryMacros.h>
#include <SWI-cpp.h>
#include <OgreMemoryMacros.h>

namespace Odin {
    class PrologAdapter {
    public:
        PrologAdapter();
        ~PrologAdapter();
        static int consult(const String& fileName);
        static int assertSomething(const String& something);
```

```
        static int retractSomething(const String& something, bool all);
        static bool unifyPredicate(const String& predicate, StringVector &args);
        static int retractAll(const String& predicate, int arity);

    private:
        static String buildFact(const String& predicate, int arity);
        static String extractPredicate(const String& pred);
        static int assertFact(const String& pred, int arity);

        static int handleQuery(PlQuery& q);
        static bool isCompound(const String& pred);
    };
}

namespace Odin {
    PrologAdapter::PrologAdapter()
    {
    }

    PrologAdapter::~PrologAdapter() {}

    int PrologAdapter::consult(const String& fileName) {
        if(!HelperKit::fileExists(fileName)) {
            String err = "error opening " + fileName;
            throw OdinException(err);
        }

        int i = 0;

        {
            PlFrame fr;

            PlTermv pt(1);
```

```
        pt[0] = fileName.c_str();
        PlQuery q = PlQuery("consult", pt);
        i = handleQuery(q);

        fr.rewind();
    }

    return i;
}

int PrologAdapter::assertSomething(const String& something) {
    int i = 0;

    {
        PlFrame fr;

        PlTermv t1(PlCompound(something.c_str()));
        PlQuery q("assert", t1);
        i = handleQuery(q);

        fr.rewind();
    }

    return i;
}

int PrologAdapter::assertFact(const String& pred, int arity) {
    String s = extractPredicate(pred);
    int a = retractAll(s, arity);
    int b = assertSomething(pred);
    return a && b;
}
```

```
bool PrologAdapter::unifyPredicate(const String& predicate, StringVector &args) {
    int b = 0;
    int max = static_cast<int>(args.size());

    {
        PlFrame fr;
        PlTermv pv(max);

        for(int i = 0; i < max; i++)
            if(args[i].size() > 0) {
                if(isCompound(args[i])) {
                    String s = args[i];
                    pv[i] = PlCompound(s.c_str());
                } else
                    pv[i] = args[i].c_str();

            }

        PlQuery q = PlQuery(predicate.c_str(), pv);
        b = handleQuery(q);

        for(int i = 0; i < max; i++)
            args[i] = pv[i];
    }

    return b != 0;
}

inline
String PrologAdapter::buildFact(const String& predicate, int arity) {
    if(!arity)
        return String(predicate);
```

```
        String buffer(predicate);
        buffer += "()";
        buffer.insert(buffer.size()-1,"X0");

        for(int i = 1; i < arity; i++) {
            char buf[5];
            sprintf(buf, ",X%d", i);
            String addition(buf);
            buffer.insert(buffer.size()-1, addition);
        }

        return buffer;
    }

    inline
    String PrologAdapter::extractPredicate(const String& pred) {
        String s(pred);
        size_t k = s.find('(');
        if (k != s.npos)
            return String(s.substr(0,k));
        return s;
    }

    inline
    bool PrologAdapter::isCompound(const String& pred) {
        size_t k = pred.find('(');
        if (k != pred.npos)
            return true;
        return false;
    }

    inline
    int PrologAdapter::retractSomething(const String& something, bool all) {
```

```
    int i = 0;

    {
        PlFrame fr;

        PlTermv t1(PlCompound(something.c_str()));

        if(all) {
            PlQuery q = PlQuery("retractall", t1);
            i = handleQuery(q);
        } else {
            PlQuery q = PlQuery("retract", t1);
            i = handleQuery(q);
        }

        fr.rewind();
    }

    return i;
}

inline
int PrologAdapter::retractAll(const String& predicate, int arity) {
    String fact = buildFact(predicate, arity);
    int i = retractSomething(fact.c_str(), true);
    return i;
}

inline
int PrologAdapter::handleQuery(PlQuery& q) {
    int i = 0;
    String err = "Prolog Exception (Possibly Syntax)";
    err.append("Nested Exception is: ");
```

```
        try {
            i = q.next_solution();
        } catch (PlTermvDomainError& ptde) {
            err.append("PlTermvDomainError ");
            err.append(ptde.name());
            throw OdinException(err);
        } catch (PlDomainError& pde) {
            err.append("PlDomainError ");
            err.append(pde.name());
            throw OdinException(err);
        } catch (PlTypeError& pte) {
            err.append("PlTypeError ");
            err.append(pte.name());
            throw OdinException(err);
        } catch(PlException& pe) {
            err.append("PlException ");
            err.append(pe.name());
            throw OdinException(err);
        }

        return i;
    }
}
```

A.2 Programa PROLOG para o Cenário do Labirinto

Nesta seção listamos duas soluções para o problema apresentado no cenário do labirinto. A primeira solução se baseia na Distância de Manhattan [41]. A segunda solução se baseia no algoritmo A* [30].

A.2.1 Código Usando a Distância de Manhattan

```
findPath(node(X, Y), node(XG, YG), []) :-
  X == XG,
```

```
  Y == YG,
  !.

findPath(node(X1, Y1), node(X2, Y2), Path) :-
  validate(X1, X2, Y1, Y2),
  getPaceX(X1,X2,PaceX,DirX),
  getPaceY(Y1,Y2,PaceY,DirY),
  doX(PathX, node(X1, Y1), node(X2, Y2), StoppedX, PaceX, DirX),
  doY(PathY, node(StoppedX, Y1), node(X2, Y2), PaceY, DirY),
  append(PathX, PathY, Path),
  !.

validate(X1, X2, _, _) :-
  X1 =\= X2.

validate(_, _, Y1, Y2) :-
  Y1 =\= Y2.

doX([],node(X,_),node(X,_),X, _, _).
doX([HEAD|TAIL],node(X1,Y1),node(X2,_),StoppedX,Pace,Direction) :-
  NewX is X1 + Pace,
  HEAD = Direction,
  doX(TAIL,node(NewX,Y1),node(X2,_),StoppedX,Pace,Direction).

doY([],node(_,Y),node(_,Y),_,_).
doY([HEAD|TAIL],node(X1,Y1),node(_,Y2),Pace,Direction) :-
  NewY is Y1 + Pace,
  HEAD = Direction,
  doY(TAIL,node(X1,NewY),node(_,Y2),Pace,Direction).

getPaceX(X1, X2, Pace, Direction) :-
  X1 < X2,
  Pace is 1,
```

```
  Direction = d3.

getPaceX(_, _, Pace, Direction) :-
  Pace is -1,
  Direction = d7.

getPaceY(Y1, Y2, Pace, Direction) :-
  Y1 < Y2,
  Pace is 1,
  Direction = d1.

getPaceY(_, _, Pace, Direction) :-
  Pace is -1,
  Direction = d5.
```

A.2.2 Código Usando o Algoritmo A-estrela

```
:- op(400,yfx,'#').

findPath(astar, From, Goal, Path) :-
  f_function(From,Goal,0,F),
  search([From#0#F#[]],Goal,S,[]),
  reverse(S,Path).

%------------UP-----------------%
up(node(X, Y), node(X, A), _) :-
  A is Y + 1,
  obstacle(node(X,A)),
  !,
  fail.
up(node(X, Y), node(X, A), Closed) :-
  A is Y + 1,
  member(node(X, A), Closed),
```

```
  !,
  fail.
up(node(X, Y), node(X, A), _) :-
  A is Y + 1.

%------------upRight-----------------%
upRight(node(X,Y), node(A, B), _) :-
  A is X + 1,
  B is Y + 1,
  obstacle(node(A,B)),
  !,
  fail.
upRight(node(X,Y), node(A, B), Closed) :-
  A is X + 1,
  B is Y + 1,
  member(node(A, B), Closed),
  !,
  fail.
upRight(node(X,Y), node(A, B), _) :-
  A is X + 1,
  B is Y + 1.

%------------right-----------------%
right(node(X,Y), node(A,Y), _) :-
  A is X + 1,
  obstacle(node(A,Y)),
  !,
  fail.
right(node(X,Y), node(A,Y), Closed) :-
  A is X + 1,
  member(node(A, Y), Closed),
  !,
  fail.
```

```
right(node(X,Y), node(A,Y), _) :-
  A is X + 1.

%------------rightDown-----------------%
rightDown(node(X,Y), node(A,B), _) :-
  A is X + 1,
  B is Y - 1,
  obstacle(node(A,B)),
  !,
  fail.
rightDown(node(X,Y), node(A,B), Closed) :-
  A is X + 1,
  B is Y - 1,
  member(node(A,B), Closed),
  !,
  fail.
rightDown(node(X,Y), node(A,B), _) :-
  A is X + 1,
  B is Y - 1.

%------------down-----------------%
down(node(X,Y), node(X,A), _) :-
  A is Y - 1,
  obstacle(node(X,A)),
  !,
  fail.
down(node(X,Y), node(X,A), Closed) :-
  A is Y - 1,
  member(node(X,A), Closed),
  !,
  fail.
down(node(X,Y), node(X,A), _) :-
  A is Y - 1.
```

```
%------------downLeft-----------------%
downLeft(node(X,Y), node(A,B), _) :-
  A is X - 1,
  B is Y - 1,
  obstacle(node(A,B)),
  !,
  fail.
downLeft(node(X,Y), node(A,B), Closed) :-
  A is X - 1,
  B is Y - 1,
  member(node(A,B), Closed),
  !,
  fail.
downLeft(node(X,Y), node(A,B), _) :-
  A is X - 1,
  B is Y - 1.

%------------left-----------------%
left(node(X,Y), node(A,Y), _) :-
  A is X - 1,
  obstacle(node(A,Y)),
  !,
  fail.
left(node(X,Y), node(A,Y), Closed) :-
  A is X - 1,
  member(node(A, Y), Closed),
  !,
  fail.
left(node(X,Y), node(A,Y), _) :-
  A is X - 1.

%------------leftUp-----------------%
```

```
leftUp(node(X,Y), node(A,B), _) :-
  A is X - 1,
  B is Y + 1,
  obstacle(node(A,B)),
  !,
  fail.
leftUp(node(X,Y), node(A,B), Closed) :-
  A is X - 1,
  B is Y + 1,
  member(node(A,B),Closed),
  !,
  fail.
leftUp(node(X,Y), node(A,B), _) :-
  A is X - 1,
  B is Y + 1.

move(A,B,Closed,d1,10) :-
  up(A,B,Closed).
move(A,B,Closed,d2,14) :-
  upRight(A,B,Closed).
move(A,B,Closed,d3,10) :-
  right(A,B,Closed).
move(A,B,Closed,d4,14) :-
  rightDown(A,B,Closed).
move(A,B,Closed,d5,10) :-
  down(A,B,Closed).
move(A,B,Closed,d6,14) :-
  downLeft(A,B,Closed).
move(A,B,Closed,d7,10) :-
  left(A,B,Closed).
move(A,B,Closed,d8,14) :-
  leftUp(A,B,Closed).
```

```
f_function(State, Goal, D, F) :-
  h_function(State, Goal, H),
  F is D + H.

h_function(node(X,Y), node(XG, YG), H) :-
  h_diagonal(X,Y,XG,YG,Diag),
  h_straight(X,Y,XG,YG,Straight),
  H is 10 * Straight - 6 * Diag.

h_straight(X,Y,FX,FY,H) :-
  dist(X,FX,DX),
  dist(Y,FY,DY),
  H is DX + DY.

h_diagonal(X,Y,FX,FY,H) :-
  dist(X, FX, DX),
  dist(Y, FY, DY),
  DX < DY,
  H is DX,
  !.

h_diagonal(_,Y,_,FY,H) :-
  dist(Y, FY, DY),
  H is DY.

dist(A,A,0) :- !. dist(A,B,D) :-
  A < B,
  DX is B - A,
  D is abs(DX),
  !.
dist(A,B,D) :-
  DX is A - B,
  D is abs(DX).
```

```
search([State#_#_#Solution|_], Goal, Solution, _) :-
  State == Goal,
  !.

search([B|R], Goal, S,InitClosed) :-
  expand(B,Children,InitClosed,Closed, Goal),
  insert_all(Children,R,Open),
  search(Open, Goal, S,Closed).

insert_all([F|R],Open1,Open3) :-
  insert(F,Open1,Open2),
  insert_all(R,Open2,Open3).
insert_all([],Open,Open).

insert(B,Open,Open) :- repeat_node(B,Open), !.
insert(B,[C|R],[B,C|R]) :- cheaper(B,C), ! . insert(B,[B1|R],[B1|S])
:- insert(B,R,S), !. insert(B,[],[B]). repeat_node(P#_#_#_,
[P#_#_#_|_]). cheaper(_#_#F1#_ , _#_#F2#_) :- F1 < F2.

expand(State#D#_#S,All_My_Children,InitClosed,Closed,Goal) :-
  append([State], InitClosed, Closed),
  findall(Child#D1#F#[Move|S],
          (move(State,Child,Closed,Move,Cost),
           D1 is D + Cost,
           f_function(Child, Goal, D1,F)
          ),
          All_My_Children
  ),
  !.

expand(State#_#_#_,[],InitClosed,Closed,_) :-
  append([State], InitClosed, Closed).
```

A.3 Programa PROLOG para o Cenário do Mundo de Wumpus

Abaixo temos um possível programa PROLOG que tenta resolver o cenário do Mundo de Wumpus. Com esse programa o agente sempre se desvia para a direita ao encontrar um obstáculo e para a frente se não houver obstáculo. Se o agente estiver no mesmo quadrado que a pepita de ouro, ele irá coletá-la. O agente consegue determinar exatamente onde está o Wumpus dependendo do número de quadrados contendo o odor do Wumpus que visitar.

```
:- dynamic([
  %Odin Declarations
  adventurer_orientation/1,
  adventurer_location/2,
  sideSquareNumber/1,
  percepts/5,
  canPunch/1,


  %Custom Declarations
  adventurer_initial_location/2,
  possible_wumpus_location/3,
  precise_wumpus_location/2,
  stench_found/3,
  number_of_stenches_found/1,
  possible_pit_location/3,
  initiated/0,
  goal/1,
  is_wumpus_dead/1,
  is_gold_found/1
]).

act(initializing):-
  negate(initiated),
```

```
  !,
  initialize.
act(Action):-
  update_map,
  do(Action).

initialize:-
  retractall(possible_wumpus_location(_,_,_)),
  retractall(stench_found(_,_,_)),
  retractall(number_of_stenches_found(_)),
  retractall(possible_pit_location(_,_,_)),
  retractall(initiated),
  retractall(goal(_)),
  assert(initiated),
  assert(goal(explore)),
  assert(number_of_stenches_found(0)),
  assert(is_wumpus_dead(no)),
  adventurer_location(X,Y),
  assert(adventurer_initial_location(X,Y)),
  assert(is_gold_found(no)).


update_map:-
  percepts(Stench,Breeze,_,_,Scream),
  adventurer_location(X,Y),
  track_wumpus_health(Scream),
  track_pit(Breeze,X,Y),
  track_wumpus(Stench,X,Y).

track_wumpus_health(yes):-
  retractall(is_wumpus_dead(_)),
  assert(is_wumpus_dead(yes)).
track_wumpus_health(no).
```

```
track_wumpus(_,_,_):-
  is_wumpus_dead(yes).s
track_wumpus(no,X,Y):-
  update_wumpus_map(no,X,Y),
  adjacent_square(X,Y,d1,X1,Y1),
  update_wumpus_map(no,X1,Y1),
  adjacent_square(X,Y,d2,X2,Y2),
  update_wumpus_map(no,X2,Y2),
  adjacent_square(X,Y,d3,X3,Y3),
  update_wumpus_map(no,X3,Y3),
  adjacent_square(X,Y,d4,X4,Y4),
  update_wumpus_map(no,X4,Y4),
  pinpoint_wumpus.
track_wumpus(yes,X,Y):-
  stench_found(_,X,Y),
  !.
track_wumpus(yes,X,Y):-
  number_of_stenches_found(Stench_Number),
  New_Stench is Stench_Number + 1,
  assert(stench_found(New_Stench,X,Y)),
  increment_number_of_stenches_found,
  update_wumpus_map(no,X,Y),
  adjacent_square(X,Y,d1,X1,Y1),
  update_wumpus_map(yes,X1,Y1),
  adjacent_square(X,Y,d2,X2,Y2),
  update_wumpus_map(yes,X2,Y2),
  adjacent_square(X,Y,d3,X3,Y3),
  update_wumpus_map(yes,X3,Y3),
  adjacent_square(X,Y,d4,X4,Y4),
  update_wumpus_map(yes,X4,Y4),
  pinpoint_wumpus.
```

```
pinpoint_wumpus:-
  is_wumpus_dead(yes).
pinpoint_wumpus:-
  number_of_stenches_found(0),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(3),
  stench_found(1,X1,Y1),
  stench_found(2,X2,Y2),
  stench_found(3,X3,Y3),
  find_common_adjacency(X1,Y1, X2,Y2, X3,Y3, Xa,Ya),
  wumpus_has_been_found(Xa,Ya),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(2),
  stench_found(1,X1,Y1),
  stench_found(2,X2,Y2),
  test_opposing(X1,Y1, X2,Y2, Xa,Ya),
  wumpus_has_been_found(Xa,Ya),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(2),
  stench_found(1,X1,Y1),
  stench_found(2,X2,Y2),
  find_possible_wumpus_squares(X1,Y1, X2,Y2, Xa,Ya, Xb,Yb),
  possible_wumpus_location(no,Xa,Ya),
  wumpus_has_been_found(Xb,Yb),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(2),
  stench_found(1,X1,Y1),
  stench_found(2,X2,Y2),
```

```
  find_possible_wumpus_squares(X1,Y1, X2,Y2, Xa,Ya, Xb,Yb),
  possible_wumpus_location(no,Xb,Yb),
  wumpus_has_been_found(Xa,Ya),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(1),
  stench_found(1,Xs,Ys),
  find_possible_wumpus_squares_one_stench(Xs,Ys, X1,Y1, X2,Y2, X3,Y3, X4,Y4),
  invalid_or_wumpus_free_square(X2,Y2),
  invalid_or_wumpus_free_square(X3,Y3),
  invalid_or_wumpus_free_square(X4,Y4),
  valid_square(X1,Y1),
  negate(possible_wumpus_location(no,X1,Y1)),
  wumpus_has_been_found(X1,Y1),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(1),
  stench_found(1,Xs,Ys),
  find_possible_wumpus_squares_one_stench(Xs,Ys, X1,Y1, X2,Y2, X3,Y3, X4,Y4),
  invalid_or_wumpus_free_square(X1,Y1),
  invalid_or_wumpus_free_square(X3,Y3),
  invalid_or_wumpus_free_square(X4,Y4),
  valid_square(X2,Y2),
  negate(possible_wumpus_location(no,X2,Y2)),
  wumpus_has_been_found(X2,Y2),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(1),
  stench_found(1,Xs,Ys),
  find_possible_wumpus_squares_one_stench(Xs,Ys, X1,Y1, X2,Y2, X3,Y3, X4,Y4),
  invalid_or_wumpus_free_square(X1,Y1),
  invalid_or_wumpus_free_square(X2,Y2),
  invalid_or_wumpus_free_square(X4,Y4),
```

```
  valid_square(X3,Y3),
  negate(possible_wumpus_location(no,X3,Y3)),
  wumpus_has_been_found(X3,Y3),
  !.
pinpoint_wumpus:-
  number_of_stenches_found(1),
  stench_found(1,Xs,Ys),
  find_possible_wumpus_squares_one_stench(Xs,Ys, X1,Y1, X2,Y2, X3,Y3, X4,Y4),
  invalid_or_wumpus_free_square(X1,Y1),
  invalid_or_wumpus_free_square(X2,Y2),
  invalid_or_wumpus_free_square(X3,Y3),
  valid_square(X4,Y4),
  negate(possible_wumpus_location(no,X4,Y4)),
  wumpus_has_been_found(X4,Y4),
  !.
pinpoint_wumpus.

valid_square(X,Y):-
  X >= 0,
  Y >= 0,
  sideSquareNumber(Number),
  X < Number,
  Y < Number.

invalid_or_wumpus_free_square(X,Y):-
  negate(valid_square(X,Y)).
invalid_or_wumpus_free_square(X,Y):-
  possible_wumpus_location(no,X,Y).

find_possible_wumpus_squares_one_stench(Xs,Ys, Xs,Y1, X2,Ys,Xs,Y3, X4,Ys):-
  Y1 is Ys + 1,
  X2 is Xs - 1,
  Y3 is Ys - 1,
```

```
  X4 is Xs + 1.

find_possible_wumpus_squares(X1,Y1, X2,Y2, X1,Y2, X2,Y1).

find_common_adjacency(X1,Y1, X2,Y2, _,_, Xa,Ya):-
  test_opposing(X1,Y1, X2,Y2, Xa,Ya),
  !.
find_common_adjacency(X1,Y1, _,_, X3,Y3, Xa,Ya):-
  test_opposing(X1,Y1, X3,Y3, Xa,Ya),
  !.
find_common_adjacency(_,_, X2,Y2, X3,Y3, Xa,Ya):-
  test_opposing(X2,Y2, X3,Y3, Xa,Ya),
  !.

test_opposing(X1,Y1, X2,Y2, X1,Ya):-
  X1 = X2,
  !,
  find_middle_number(Y1,Y2,Ya).
test_opposing(X1,Y1, X2,Y2, Xa,Y1):-
  Y1 = Y2,
  !,
  find_middle_number(X1,X2,Xa).

find_middle_number(Point1,Point2,Middle):-
  Point2 >= Point1,
  !,
  Middle is Point1 + ((Point2 - Point1) // 2).
find_middle_number(Point1,Point2,Middle):-
  Point2 < Point1,
  !,
  Middle is Point2 + ((Point1 - Point2) // 2).

wumpus_has_been_found(X,Y):-
```

```
  retractall(precise_wumpus_location(_,_)),
  assert(precise_wumpus_location(X,Y)).

grab_adjacencies(X,Y, X1,Y1, X2,Y2, X3,Y3, X4,Y4):-
  adjacent_square(X,Y,d1,X1,Y1),
  adjacent_square(X,Y,d2,X2,Y2),
  adjacent_square(X,Y,d3,X3,Y3),
  adjacent_square(X,Y,d4,X4,Y4).

track_pit(no,X,Y):-
  update_pit_map(no,X,Y),
  adjacent_square(X,Y,d1,X1,Y1),
  update_pit_map(no,X1,Y1),
  adjacent_square(X,Y,d2,X2,Y2),
  update_pit_map(no,X2,Y2),
  adjacent_square(X,Y,d3,X3,Y3),
  update_pit_map(no,X3,Y3),
  adjacent_square(X,Y,d4,X4,Y4),
  update_pit_map(no,X4,Y4).
track_pit(yes,X,Y):-
  update_pit_map(no,X,Y),
  adjacent_square(X,Y,d1,X1,Y1),
  update_pit_map(yes,X1,Y1),
  adjacent_square(X,Y,d2,X2,Y2),
  update_pit_map(yes,X2,Y2),
  adjacent_square(X,Y,d3,X3,Y3),
  update_pit_map(yes,X3,Y3),
  adjacent_square(X,Y,d4,X4,Y4),
  update_pit_map(yes,X4,Y4).

update_wumpus_map(no,X,Y):-
  retractall(possible_wumpus_location(_,X,Y)),
  assert(possible_wumpus_location(no,X,Y)),
```

```
  !.
update_wumpus_map(yes,X,Y):-
  possible_wumpus_location(no,X,Y),
  !.
update_wumpus_map(yes,X,Y):-
  retractall(possible_wumpus_location(_,X,Y)),
  assert(possible_wumpus_location(yes,X,Y)).


update_pit_map(no,X,Y):-
  retractall(possible_pit_location(_,X,Y)),
  assert(possible_pit_location(no,X,Y)),
  !.
update_pit_map(yes,X,Y):-
  possible_pit_location(no,X,Y),
  !.
update_pit_map(yes,X,Y):-
  retractall(possible_pit_location(_,X,Y)),
  assert(possible_pit_location(yes,X,Y)).

adjacent_square(X,Y,d1,Nx,Y):- Nx is X+1.
adjacent_square(X,Y,d2,X,Ny):- Ny is Y+1.
adjacent_square(X,Y,d3,Nx,Y):- Nx is X-1.
adjacent_square(X,Y,d4,X,Ny):- Ny is Y-1.

do(climb):-
  adventurer_location(X,Y),
  adventurer_initial_location(X,Y),
  is_gold_found(yes),
  !.
do(right):-
  goal(explore),
  adventurer_location(X,Y),
```

```
  adventurer_orientation(Orientation),
  adjacent_square(X,Y,Orientation,X1,X2),
  negate(valid_square(X1,X2)),
  !.
do(pick):-
  goal(explore),
  percepts(_,_,yes,_,_),
  assert_gold_found(yes),
  !.
do(punch):-
  adventurer_location(X,Y),
  adventurer_orientation(O),
  adjacent_square(X,Y,O,X1,X2),
  precise_wumpus_location(X1,X2),
  is_wumpus_dead(no),
  !.
do(right):-
  goal(explore),
  adventurer_location(X,Y),
  adventurer_orientation(O),
  adjacent_square(X,Y,O,X1,X2),
  possible_wumpus_location(yes,X1,X2),
  is_wumpus_dead(no),
  !.
do(right):-
  goal(explore),
  adventurer_location(X,Y),
  adventurer_orientation(O),
  adjacent_square(X,Y,O,X1,X2),
  possible_pit_location(yes,X1,X2),
  !.
do(move):-
  goal(explore).
```

```
negate(Fact) :-
  Fact,
  !,
  fail.
negate(_).

increment_number_of_stenches_found:-
  number_of_stenches_found(I),
  J is I + 1,
  retractall(number_of_stenches_found(_)),
  assert(number_of_stenches_found(J)).

assert_goal(Goal):-
  retractall(goal(_)),
  assert(goal(Goal)).

assert_gold_found(no):-
  retractall(is_gold_found(_)),
  assert(is_gold_found(no)).
assert_gold_found(yes):-
  retractall(is_gold_found(_)),
  assert(is_gold_found(yes)).
```

Referências Bibliográficas

[1] *A Debate on Teaching Computing Science*, Commun. ACM **32** (1989), no. 12, 1397–1414. 10

[2] *SWI-Prolog 5.6.45 Reference Manual*, `http://gollem.science.uva.nl/SWI-Prolog/Manual/`, 2007, [Online; acessado 20-Novembro-2007]. 40

[3] *The M.U.P.P.E.T.S. Project @ RIT*, `http://muppets.rit.edu`, 2007, [Online; acessado 20-Novembro-2007]. xiii, 9, 16, 18

[4] David H. Ahl, *MORE BASIC Computer Games*, pp. 178–182, Workman Publishing, 1979. 36

[5] Robert St. Amant and R. Michael Young, *Links: Artificial Intelligence and Interactive Entertainment*, Intelligence **12** (2001), no. 2, 17–19. 6

[6] Mazin Assanie, Benjamin Bachelor, Nathan Benninghoff, Syed Enam, Bradley Jones, Alex Kerfoot, Colin Lauver, Brian Magerko, Jeff Sheiman, Devvan Stokes, and Scott Wallace, *A Test Bed for Developing Intelligent Synthetic Characters*, In Spring Symposium on Artificial Intelligence and Interactive Entertainment, AAAI, 2002. 9

[7] Jonathan Baron, *Glory and Shame: Powerful Psychology in Multiplayer Online Games*, Proceedings of the Game Developers Conference, IGDA, 1999, [Online; acessado 20-Novembro-2007]. 14

[8] Jon Barwise and John Etchemendy, *Tarski's world 3.0: Including the macintosh tm program (center for the study of language and information - lecture notes)*, Center for the Study of Language and Information/SRI, 1991. 24

[9] Kevin J. Bierre and Andrew M. Phelps, *The Use of MUPPETS in an Introductory Java Programming Course*, CITC5 '04: Proceedings of the 5th conference on Information technology education (New York, NY, USA), ACM Press, 2004, pp. 122–127. xiii, 15, 18, 19, 20

[10] Kevin J. Bierre, Phil Ventura, Andrew M. Phelps, and Christopher Egert, *Motivating OOP by Blowing Things up: an Exercise in Cooperation and Competition in an Introductory Java Programming Course*, SIGCSE Bull. **38** (2006), no. 1, 354–358. iii, v

[11] Bruce M. Blumberg and Tinsley A. Galyean, *Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments*, SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, ACM Press, 1995, pp. 47–54. 1

[12] C. Bouras, V. Triantafillou, and T. Tsiatsos, *Aspects of Collaborative Learning Environment Using Distributed Virtual Environments*, ED-MEDIA '01: Proceedings of the 8th World Conference on Educational Multimedia, Hypermedia & Telecommunications, AACE, 2001. 11

[13] Paul Brna, *Prolog Programming: a First Course*, `http://computing.unn.ac.uk/staff/cgpb4/prologbook/`, 2001, [Online; acessado 20-Novembro-2007]. 2, 28, 29, 31, 33, 34, 35

[14] Mike Brockington and Scott Greig, *GDC 2003: Neverwinter Nights Client/Server Postmortem: How I Learned To Stop Worrying And Love The Magic Missile*, `http://www.gamasutra.com/gdc2003/features/20030306/brockington_02.htm`, 2003, [Online; acessado 20-Novembro-2007]. 6

[15] Elizabeth F. Churchill, David N. Snowdon, and Alan J. Munro, *Collaborative Virtual Environments*, p. 3, Springer, 2001. 11

[16] Ron Coleman, Mary Krembs, Alan Labouseur, and Jim Weir, *Game design & programming concentration within the computer science curriculum*, SIGCSE Bull. **37** (2005), no. 1, 545–550. 24

[17] Matthew J. Conway, *Alice: Easy-to-Learn 3D Scripting for Novices*, Tese de Doutorado, Faculty of the School of Engineering and Applied Science at the University of Virginia, 1997. 21, 22

[18] Stephen A. Cook, *The Complexity of Theorem-Proving Procedures*, STOC '71: Proceedings of the third annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1971, pp. 151–158. 28

[19] Stephen Cooper, Wanda Dann, and Randy Pausch, *Alice: a 3-D Tool for Introductory Programming Concepts*, CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges (Mahwah, NJ, USA), Consortium for Computing Sciences in Colleges, 2000, pp. 107–116. 1, 22

[20] _____, *Teaching Objects-First in Introductory Computer Science*, SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, 2003, pp. 191–195. 23

[21] Filipe Correa Lima da Silva and Flávio Soares Corrêa da Silva, *Um Ambiente Virtual Baseado em Jogos para o Aprendizado de Inteligência Artificial*, SBGames 2005: Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital (São Paulo, SP, Brasil), USP, 2005. 72

[22] _____, *A Game-based Animation Tool to Support the Teaching of Formal Reasoning*, SBGames 2006: DIGITAL PROCEEDINGS of the V Brazilian Symposium on Computer Games and Digital Entertainment (Recife, PE, Brazil), Porto Digital, 2006. 72

[23] Wanda Dann, Stephen Cooper, and Randy Pausch, *Making the Connection: Programming with Animated Small World*, ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE-conference on Innovation and technology in computer science education (New York, NY, USA), ACM Press, 2000, pp. 41–44. 23

[24] _____, *Using Visualization to Teach Novices Recursion*, ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education (New York, NY, USA), ACM Press, 2001, pp. 109–112. 23

[25] Wanda Dann, Toby Dragon, Stephen Cooper, Kevin Dietzler, Kathleen Ryan, and Randy Pausch, *Objects: Visualization of Behavior and State*, SIGCSE Bull. **35** (2003), no. 3, 84–88. 22

[26] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice in C*, Addison-Wesley Professional, Boston, MA, USA, 1995. 46

[27] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor, *On visible surface generation by a priori tree structures*, SIGGRAPH Comput. Graph. **14** (1980), no. 3, 124–133. 48

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 46

[29] Astrid Glende, *Agent Design to Pass Computer Games*, ACMSE'04: Proceedings of the 42nd annual ACM Southeast regional conference, ACM Press, 2004, pp. 414–415. 1

[30] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics **SSC-4** (1968), no. 2, 100–107. 60, 81

[31] Alfred Horn, *On Sentences Which are True of Direct Unions of Algebras*, The Journal of Symbolic Logic **16** (1951), no. 1, 14–21. 29

[32] IGDA, *The 2005 Report of the IGDA's Artificial Intelligence Interface Standards Committee*, `http://www.igda.org/ai/report-2005/report-2005.html`, 2005, [Online; acessado 20-Novembro-2007]. 6

[33] Robert Ingpen and Michael Page, *Encyclopedia of Things That Never Were: Creatures, Places, and People*, Studio, 1998. 38

[34] Tony Jenkins, *The Motivation of Students of Programming*, ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education (New York, NY, USA), ACM Press, 2001, pp. 53–56. 10

[35] ———, *On the Difficulty of Learning to Program*, 3rd annual Conference of LTSN-ICS (Loughbourgh), 2002. 10

[36] Randolph M. Jones, *Design and implementation of computer games: a capstone course for undergraduate computer science education*, SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, 2000, pp. 260–264. 24

[37] Gal A. Kaminka, Manuela M. Veloso, Steve Schaffer, Chris Sollitto, Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada, *GameBots: a Flexible Testbed for Multiagent Team Research*, Communications of the ACM **45** (2002), no. 1. 1

[38] Richard M. Karp, *Reducibility among Combinatorial Problems*, Complexity of Computer Computations (R. E. Miller and J. W. Thatcher, eds.), Plenum Press, New York, NY, USA, 1972, pp. 85–103. 28

[39] Brian W. Kernighan and Dennis Ritchie, *The c programming language*, Prentice-Hall, 1978. 33

[40] Robert A. Kowalski, *The Early Years of Logic Programming*, Commun. ACM **31** (1988), no. 1, 38–43. 28

[41] Eugene F. Krause, *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*, Dover Publications, New York, NY, USA, 1986. 59, 81

[42] John E. Laird, *Research in human-level AI using computer games*, Communications of the ACM **45** (2002), no. 1, 32–35. 1, 9

[43] John E. Laird, Allen Newell, and Paul S. Rosenbloom, *SOAR: an architecture for general intelligence*, Artificial Intelligence **33** (1987), no. 1, 1–64. 7

[44] John E. Laird and Michael van Lent, *Developing an Artificial Intelligence Engine*, Proceedings of the Game Developers Conference, IGDA, 1999, pp. 577–588. 7

[45] ———, *Human-Level AI's Killer Application: Interactive Computer Games*, Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI Press / The MIT Press, 2000, pp. 1171–1178. 1, 5, 6

[46] Jill F. Lehman, John E. Laird, and Paul S. Rosenbloom, *A Gentle Introduction to Soar An Architecture For Human Cognition: 2006 Update*, `http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf`, 2006, [Online; acessado 20-Novembro-2007]. 7

[47] Brian Magerko, John E. Laird, Mazin Assanie, Alex Kerfoot, and Devvan Stokes, *AI Characters and Directors for Interactive Computer Games*, Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference, AAAI Press, 2004. 1, 9

[48] John McCarthy, *Programs with Common Sense*, Proceedings of the Teddington Conference on the Mechanization of Thought Processes (London), Her Majesty's Stationary Office, 1959, pp. 75–91. 27

[49] Gildas Ménier, *3D (Realtime) and Prolog*, `http://www.arsaniit.com/vp_tools/opengl.htm`, 2007, [Online; acessado 20-Novembro-2007]. 39

[50] Tomas Moller, Eric Haines, and Tomas Akenine-Moller, *Real-Time Rendering*, AK Peters, Ltd., Wellesley, MA, USA, 2002. 46

[51] Barbara Moskal, Deborah Lurie, and Stephen Cooper, *Evaluating the effectiveness of a new instructional approach*, SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education (New York, NY, USA), ACM Press, 2004, pp. 75–79. 23

[52] Lasse Natvig and Steinar Line, *Age of computers: Game-Based Teaching of Computer Fundamentals*, ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education (New York, NY, USA), ACM Press, 2004, pp. 107–111. 24

[53] Allen Newell, *Unified theories of cognition*, Harvard University Press, Cambridge, MA, USA, 1994. 7

[54] Wizards of The Coast, *Dungeons and Dragons*, `http://www.wizards.com/dnd`, 2007, [Online; acessado 20-Novembro-2007]. 11

[55] Deborah Perelman, *An Academic Asks: Is Computer Science Dead?*, `http://www.eweek.com/article2/0,1895,2104047,00.asp`, 2007, [Online; acessado 20-Novembro-2007]. iii, v, 1

[56] Andrew M. Phelps, Kevin J. Bierre, and David M. Parks, *MUPPETS: Multi-User Programming Pedagogy for Enhancing Traditional Study*, CITC4 '03: Proceedings of the 4th Conference on Information Technology Curriculum (New York, NY, USA), ACM Press, 2003, pp. 100–105. 1, 17, 20, 27

[57] Andrew M. Phelps, Christopher A. Egert, Kevin J. Bierre, and David M. Parks, *An Open-Source CVE for Programming Education: a Case Study*, SIGGRAPH '05: ACM SIGGRAPH 2005 Courses (New York, NY, USA), ACM Press, 2005. xiii, 21

[58] Andrew M. Phelps and David M. Parks, *Fun and Games: Multi-Language Development*, ACM Queue **1** (2004), no. 10, 46–56. 18

[59] Elaine M. Raybourn, *Just How Important is 3D for CVEs?*, CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments (New York, NY, USA), ACM Press, 2002, pp. 95–96. 11

[60] Howard Rheingold, *The Virtual Community*, MIT Press, 1992, [Online; acessado 20-Novembro-2007]. 13

[61] J. A. Robinson, *A machine-oriented logic based on the resolution principle*, J. ACM **12** (1965), no. 1, 23–41. 30

[62] Stuart J. Russel and Peter Norvig, *Artifical Intelligence: A Modern Approach*, Prentice Hall, 1995. 5, 36, 72

[63] Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA, USA, 1986. 35

[64] Jan van der Steen, *Go Games, Go Information and Go Study Tools*, `http://gobase.org/`, 2007, [Online; acessado 20-Novembro-2007]. 51, 73

[65] Maarten H. van Emden and Robert A. Kowalski, *The Semantics of Predicate Logic as a Programming Language*, J. ACM **23** (1976), no. 4, 733–742. 27

[66] Michael van Lent and John E. Laird, *Learning procedural knowledge through observation*, K-CAP 2001: Proceedings of the international conference on Knowledge capture (New York, NY, USA), ACM Press, 2001, pp. 179–186. 9

[67] Michael van Lent, John E. Laird, Josh Buckman, Joe Hartford, Steve Houchard, Kurt Steinkraus, and Russ Tedrake, *Intelligent agents in computer games*, AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative Applications of Artificial Intelligence (Menlo Park, CA, USA), American Association for Artificial Intelligence, 1999, pp. 929–930. 1, 9

[68] Jay Vegso, *Interest in CS as a Major Drops Among Incoming Freshmen*, `http://www.cra.org/CRN/articles/may05/vegso`, 2005, [Online; acessado 20-Novembro-2007]. iii, v

[69] Jan Wielemaker, *An Overview of the SWI-Prolog Programming Environment*, Proceedings of the 13th International Workshop on Logic Programming Environments (Heverlee, Belgium) (Fred Mesnard and Alexander Serebenik, eds.), Katholieke Universiteit Leuven, 2003, pp. 1–16. 40

[70] R. Michael Young, *An Overview of the Mimesis Architecture: Integrating Intelligent Narrative Control into an Existing Gaming Environment*, Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment, AAAI Press (2001), AAAI Press, 2001. 1